سلسلة تعلم Angular بالعربي

RXJS In Angular

Include Angular HttpClient

المؤلف

فيصل الفهد

2022

RxJS in ANGULAR

&

Angular HttpClient

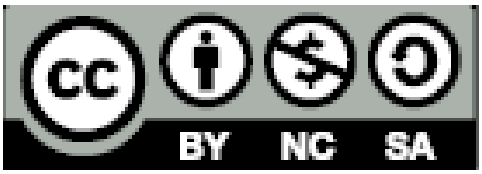
هذا الكتاب جزء من سلسلة تعلم Angular بالعربي

تم انتاج هذا العمل في عام

2022

المؤلف

فيصل الفهد



هداء

إلى أطهر قلبين في حياتي... والديَّ العزيزين.

إلى من شاركني السرَّاء والضراء، ولم أرها عابسة يومًا..... زوجتي المخلصة.

إلى من أتشوَّق لأن أرى مستقبلهما المشرق بإذن الله..... ابنائي وبناتي.

إلى جميع متلهف للعلم ويتوق للمعرفة

أهديكم هذا الكتاب المتواضع، وأدعو الله أن يحوز إعجابكم.

المؤلف

مقدمة السلسلة

اللهم علمنا ما ينفعنا وانفعنا بما علمتنا إنك انت العليم الحكيم..

أضع بين إياديكم أحبتي وأخوتي مطوري الويب وبالتحديد مطوري Front End أول سلسلة عربية تتكلم عن إطار عمل Angular ، حيث تقوم فكرة هذه السلسلة على حصر جميع الميزات التي يُقدمها Angular ومن ثم في كل كتاب يتم أخذ ميزة او ميزتين والإبحار فيها بشكل مستقل محاولا بذلك تغطية أغلب وأهم جوانبها مع الشرح المفصل والأمثلة المتعددة.

ومن هذا المنطلق يجب التنويه أن هذه السلسلة ليست للمبتدئين في البرمجة، وانما لابد ان تمتلك عزيزي المتعلم على الأقل أساسيات أركان تطوير الويب الثلاثة HTML-CSS-JavaScript ومن ثم لابد ان تمتلك اساسيات Typescript لأن إطار عمل Angular يعتمد على هذه التقنية، ولا يخفى على كل مطور واجهات أمامية أهمية Typescript والتي من وجهة نظري أرى انه يجب على كل مطور ويب تعلمها واتقانها، ونكتفي ان نقول ان Deno.js وهي اللغة الجديدة من Node.js أصبحت تدعم هذه التقنية بشكل كامل، لذلك انصح كل مطور عربي بتعلم هذه التقنية ومحاولة الإلمام بجوانبها وكيفية الاستفادة منها في إطار عمل Angular بشكل خاص ولتطوير الويب بشكل عام.

ولا يخفى عليك عزيزي المتعلم أن الكمال لله ولا يخلوا عمل من الأخطاء فالخطأ وارد والتقصير موجود، لذلك في حال وجدت أي تقصير او أي خطأ في هذه السلسلة او أردت تقديم أي اقتراح لتطوير هذه السلسلة فلا تتردد بمراسلتي على البريد الإلكتروني الذي سوف اضعه بأسفل هذه الصفحة.

أسأل الله العلي العظيم أن ينفع به وان يتقبله عملاً خالصا لوجهه سبحانه.

وأخيرا ولا تنسوني أخوتي من صالح دعائكم.

المؤلف

فيصل الفهد

Faisal.alfahd.ksa@gmail.com

جدول المحتويات

مقدمة الكتاب

في هذا الكتاب سوف نبحر وبأدق التفاصيل في Reactive Programming من خلال مكتبة RxJS وAngular، مع العلم ان تقنيات Reactive Programming ليست موجة إلى لغة الJavaScript فقط فلها مكتبات أخرى للغات متعددة مثل Java او c# او Swift .. الخ، وتستطيع مشاهدة اللغات التي تدعمها هذه التقنية من خلال هذا الرابط (https://reactivex.io)، وما يهمنا من هذه التقنية هو مكتبة RxJS وهي الموجهة إلى لغة الجافا سكريبت لذلك نستطيع استخدام هذه المكتبة مع لغة الجافا سكريبت مباشرة او مع أطر العمل الثلاثة المشهورة وهي (Angular – React – Vue)، وفي هذا الكتاب سوف نتطرق إلى استخدام هذه المكتبة مع Angular، مع العلم ان جميع المفاهيم واحدة وأغلب طرق الاستخدام متشابهة، وليس هذا فحسب بل نستطيع ان نقول ان المفاهيم واحدة لجميع لغات البرمجة مع اختلافات بسيطة في طريقة كتابة الدوال او ما يسمى Syntax بين اللغات المختلفة.

ولن نكتفي في هذا الكتاب بالتطرق إلى RxJS وانما سوف نشرح Angular HttpClient وهي الطريقة التي توفرها الAngular لاتصال HTTP لجلب او اضافة او تعديل او حذف البيانات من الخادم، لذلك قمت بتخصيص الفصل الأول من هذا الكتاب لشرح مفاهيم أساسية في الشبكات واتصالات HTTP مثال الخادم والعميل ودوال الHTTP وايضاً مفاهيم Json والAPI وغيره الكثير، بحيث نتطرق إلى هذه المفاهيم بشكل سريع بما يخدمنا في هذا الكتاب دون الخوض بتفاصيل هذه التقنيات، وفي الفصل الأخير تم تخصيصه لشرح Angular HttpClient.

المؤلف

فيصل الفهد

Faisal.alfahd.ksa@gmail.com

الفصل الأول

مدخل إلى بعض مفاهيم الشبكات وتقنيات الويب

1- مقدمة:

كما قلنا في مقدمة هذا الكتاب اننا وقبل البدء بالغوص بميزة Angular HttpClient (الفصل الأخير من هذا الكتاب) يجب قبلها ان نفهم بعض التقنيات المهمة، مثل ما هو الخادم server والعميل client وايضاً ما هو Http اصلاً، بالإضافة الى تقنية API وJson وغيرها من المفاهيم الأخرى.

وبطبيعة الحال لن نتعمق في تفاصيل هذه التقنيات لأن بعضها قد يحتاج إلى فصل مستقل ان لم يكن كتاب كامل، وانما سنقدم مدخل ومقدمة لكل تقنية ونُركز على ما نحتاجه وينفعنا لكي نتعلم ميزة HttpClient.

2- Server & Client:

كما اشرنا سابقا سوف نقوم بإعطاء مقدمة بسيطة ومدخل إلى هذه التقنيات، بحيث نُعطي الأساسيات التي تساعدنا على فهم واستيعاب HttpClient.

ومن ناحية تقنية الخادم والعميل او ما تُسمى Server & Client فهي من اهم مفاهيم شبكات الحاسب والتي أدت إلى انتشار هذه الشبكات وتعدد تطبيقاتها وخدماتها التي تقدمها للمستخدمين، فمثلاً عندما نقوم بفتح جهاز الحاسب او جهاز الموبايل او أي جهاز آخر ونقوم بالدخول على المتصفح كأن يكون Google Chrome او Safari او أي متصفح آخر وقمنا بطلب موقع وليكن محرك البحث الشهير google.com، فإننا بهذه الحالة اصبحنا عملاء Clients وقمنا بطلب خدمة وهي استعراض محرك البحث google.com وهذا الموقع بجميع ما يحتويه من ملفات ضرورية لتشغيله موجودة على جهاز حاسب آخر في منطقة معينة وهذا الجهاز يُسمى خادم Server لأنه قام بخدمة الأجهزة العميلة Clients، بشرط وجود شبكة تربط بين جهاز الخادم (السيرفر) والعميل كأن تكون شبكة انترنت او أي شبكة حاسب أُخرى.

ما تم ذكره سابقاً يُعتبر مثال بسيط لفهم وتوضيح هذه التقنيات المهمة، وإلا نستطيع ان نتصل بالسيرفر بطرق عديدة ومختلفة، وعلى سبيل المثال لا الحصر ممكن ان تكون تطبيق او لعبة.

وبنفس الوقت تختلف وتتعدد الخدمات التي يقدمها السيرفر فممكن ان يقدم سيرفر واحد مجموعة من الخدمات مثل خدمة البريد الإلكتروني او خدمة تخزين قاعدة البيانات او التطبيق الذي يتعامل مع قاعدة البيانات او تخزين ملفات الصور والوسائط الأخرى او ملفات الموقع الضرورية لعرض الموقع في جهاز العميل،...الخ، هذا من جهة ومن جهة أخرى ممكن أن يكون هنالك جهاز سيرفر لكل خدمة وبتم ربط هذه الأجهزة مع بعضها البعض بطرق معينة.

والجزء المهم الذي يجب ان نعلمه ان هنالك لغات برمجة خاصة ببرمجة تطبيقات تعمل على السيرفر وتمتلك القدرة على اجراء جميع العمليات على قاعدة البيانات سواء إضافة او حذف او تعديل او جلب البيانات من قواعد البيانات مثل لغة C# متمثلة بإطار العمل ASP.NET او لغة Python متمثلة بإطار العمل Django او لغة Java متمثلة بإطار العمل Spring Boot، وغيرهم الكثير، فهذه اللغات نستطيع عن طريقها برمجة تطبيقات تتعامل مع قاعدة البيانات او مع ملفات الوسائط على السيرفر، بصيغة أخرى نستطيع عن طريقها برمجة جميع الخدمات التي يقدمها السيرفر – أغلب هذه

الخدمات تكون جاهزة ما عدا قاعدة البيانات والعمليات عليها وملفات الوسائط – وهذا النوع من البرمجة جرى العُرف تسميته برمجة الBack-End.

وعلى الجهة المقابلة وبالتحديد في عالم الويب Web هنالك لغات برمجة لا نستطيع عن طريقها التعامل مع السيرفر وبالتحديد قاعدة البيانات او ملفات الوسائط المخزنة في السيرفر، مثل لغة JavaScript فهذه اللغة نستطيع عن طريقها برمجة الواجهات الامامية للمستخدم مثل برمجة واجهة الموقع الذي يتعامل المستخدم معه مباشرة مثل الازرار ومربعات النص، ...الخ، او عرض البيانات القادمة من السيرفر بطرق جميلة، وهذا النوع من البرمجة يُسمى برمجة Front-End لأن أوامر هذه اللغة لا يفهما إلا المتصفح فقط. (يوجد نسخة من JavaScript تم تعريفها على انها Runtime Environment تسمى Node js ونستطيع عن طريقها برمجة Back-End).

وبنفس الوقت تستطيع الJavaScript التواصل مع السيرفر عن طريق Http ودوال API، حيث عن طريق هاتين التقنيتين يتم التواصل بين التطبيق الذي تم برمجته في Back-End مع التطبيق الذي تم برمجته في Front-End، مع العمل ان المطور الذي يُجيد برمجة Back-End & Front-End يُسمى Full Stack Developer (وهو مستقبل الويب).

ملاحظة: بما ان إطار عمل Angular هو إطار عمل للJavaScript لذلك تعتبر التطبيقات التي يتم برمجتها عن طريقه من ضمن برمجة Front-End، وهذا يشمل أي تطبيق تم برمجته عن طريق Angular سواء كان Web او تطبيق موبايل عن طريق Ionic او تطبيقات سطح المكتب عن طريق Electron.

ملاحظة: هنالك بعض اللغات يمكن استخدامها للتعامل مع Front-End وBack-End مثل لغة Java حيث يتم استخدامها برمجة تطبيقات Android وهي بذلك أصبحت Front-End وايضاً Back-End كما أشرت سابقاً، وهنالك لغة Swift حيث يتم استخدامها كFront-End لبرمجة تطبيقات iOS(IPhone) وايضاً لبرمجة Back-End، وهنالك ايضاً C# لبرمجة تطبيقات سطح المكتب او حتى برمجة واجهات الويب باستخدام إطار العمل الرائع Blazor، وايضاً برمجة Back-End.

وهنالك نقطة مهمة تجب الاشارة إليها عند تطرقنا لمفاهيم الخادم والعميل وهي تقنيات الحوسبة السحابية وهذه التقنيات تُقدم لنا جميع الخدمات مع السيرفر بدون تكبد عناء شراء سيرفر جديد او صيانته وجميع الأمور الأخرى، وانما نستطيع ان نأخذ مساحة وقدرة معالجة وعدد المستخدمين الذي يستطيع ان يخدمهم هذا السيرفر وغيرها من جميع الخدمات الأخرى كقواعد البيانات والبريد الإلكتروني، ...الخ على قدر حاجتنا، وبذلك سهلة لنا بناء جميع الخدمات من الصفر، ومن الشركات التي تقدم هذا النوع من الخدمات شركة Google متمثلة في Google Cloud او شركة Microsoft متمثلة في Azure، ...الخ.

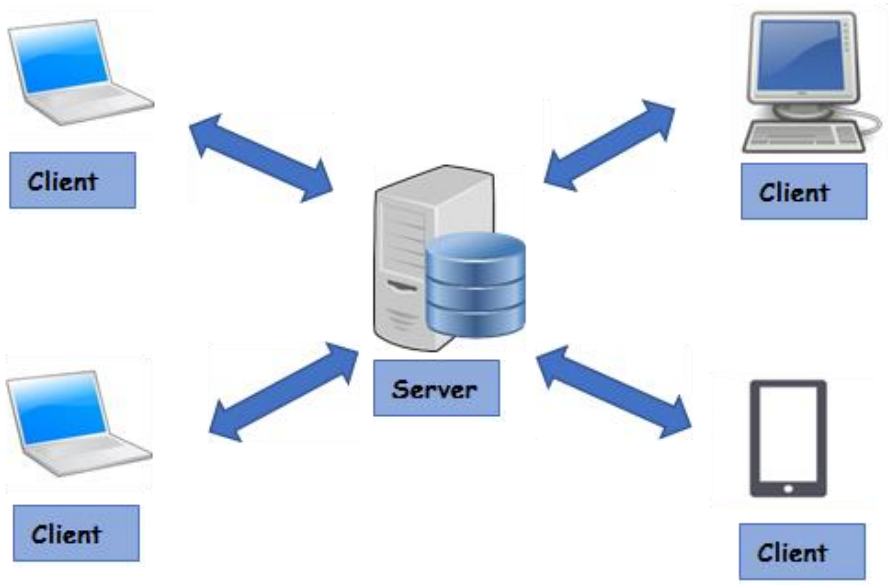
وحقيقة التحدث في مفاهيم وتقنيات الحوسبة السحابية ذو شجون وممتع ولكن ليس هنا المقام لشرحه ولذلك سنكتفي بما أشرنا له سابقاً.

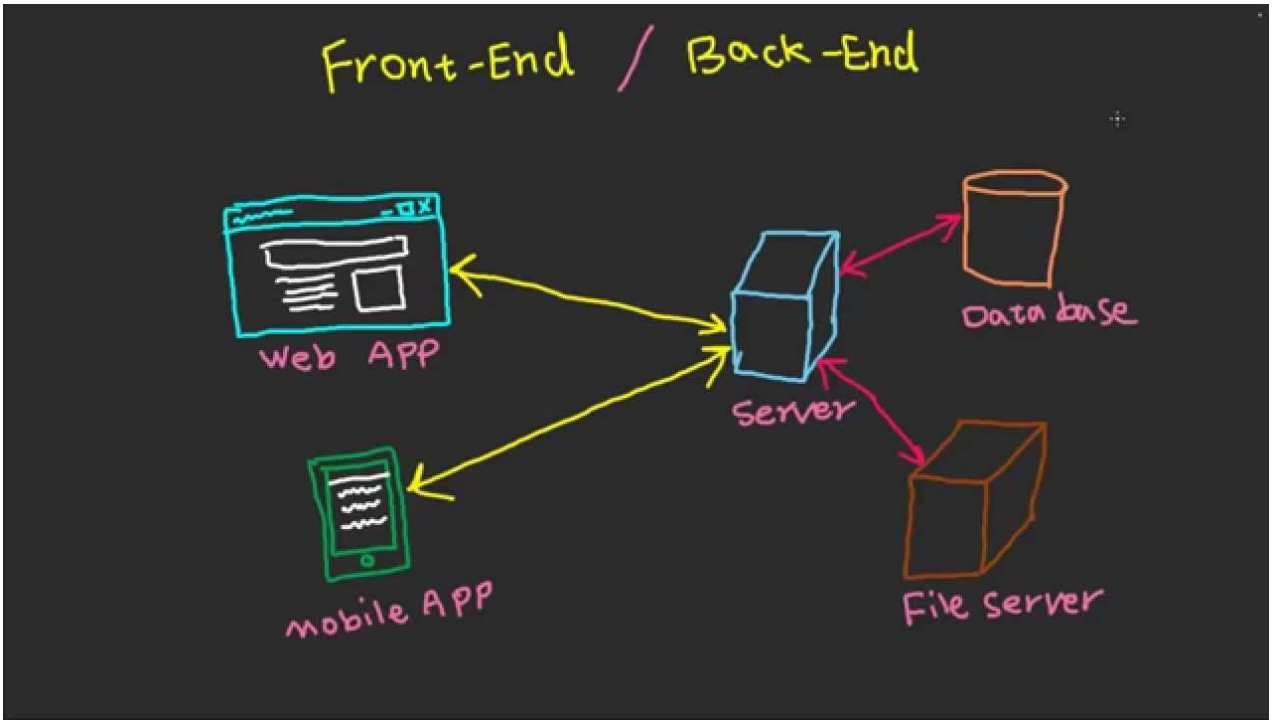
وأخيراً وبالمختصر نستطيع ان نقول:

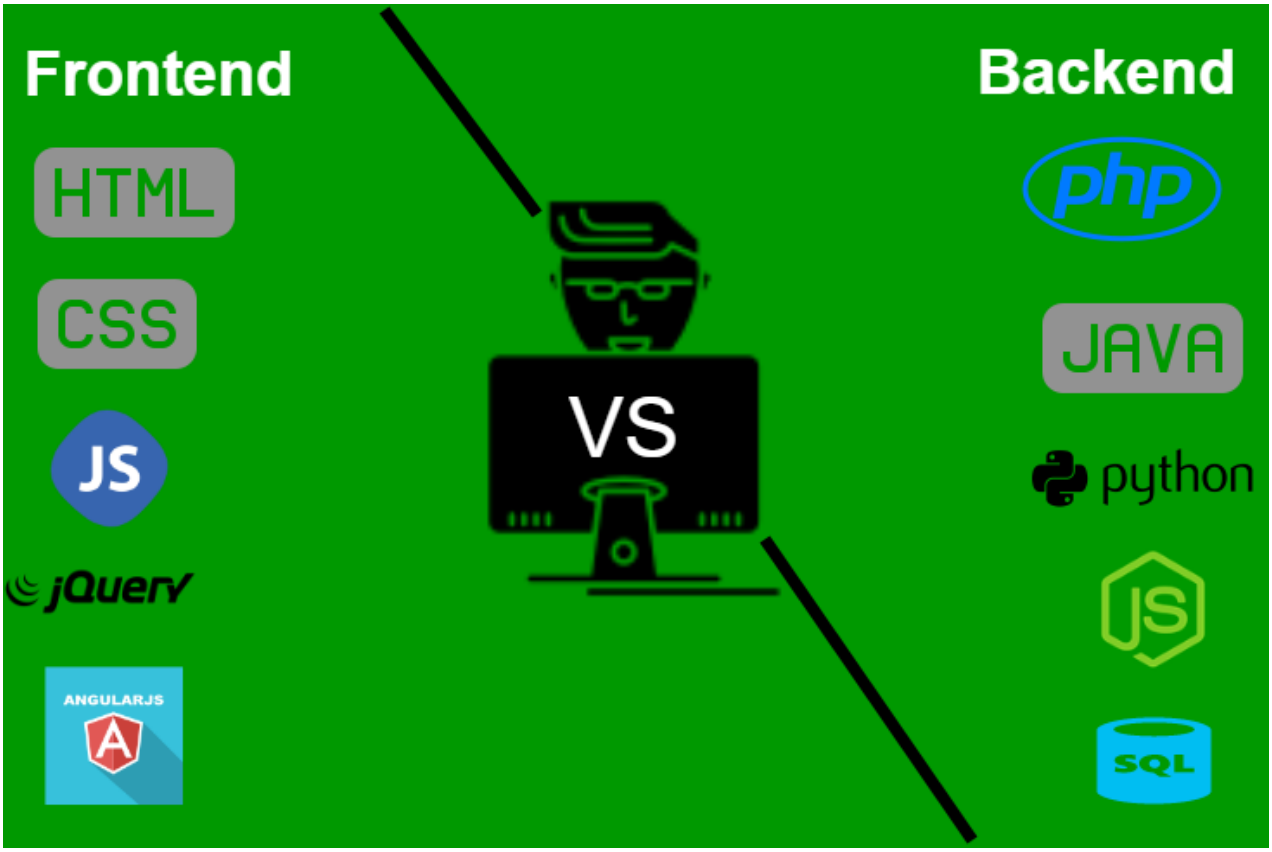
- ان أي تطبيق يعمل على السيرفر يُسمى Back-End واي تطبيق يعمل على Client يُسمى Front-End.

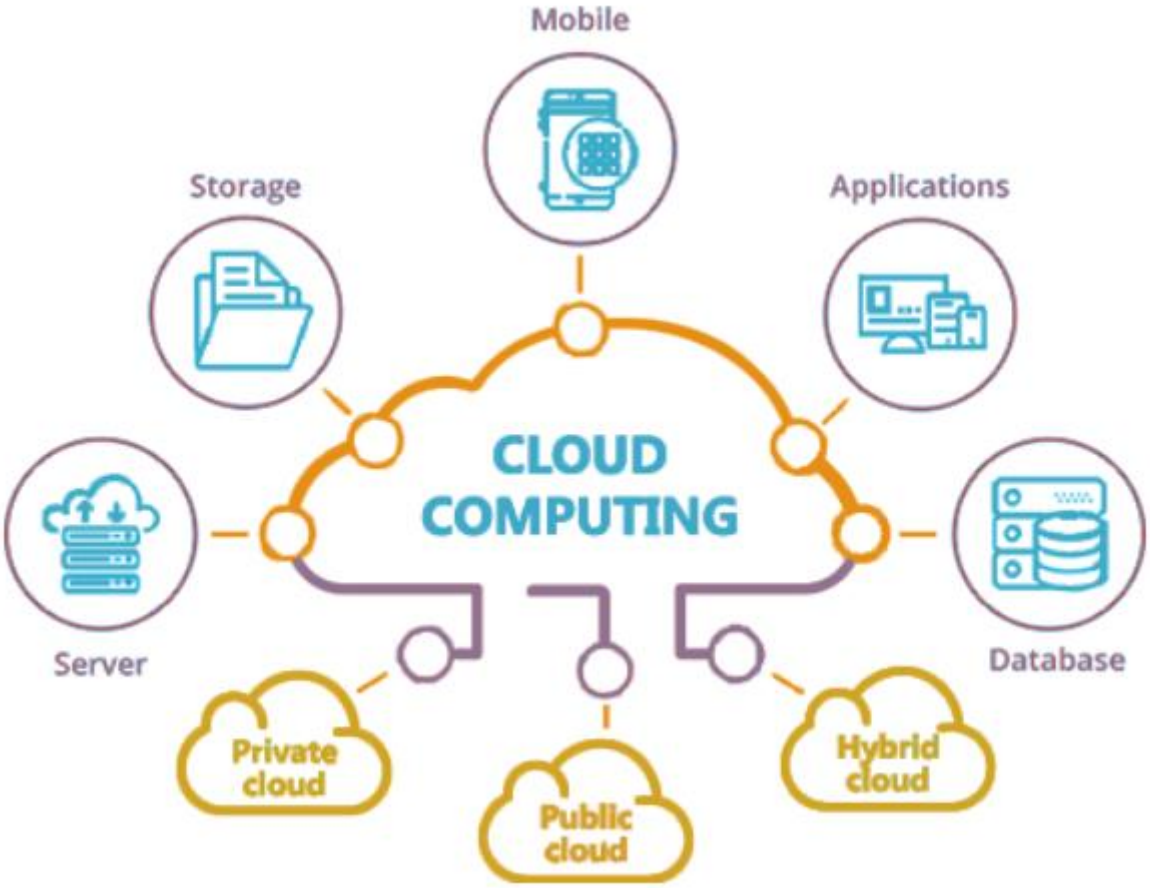
- في الغالب يتم تخزين قواعد البيانات وملفات الوسائط في السيرفر، وفي أجهزة العملاء Clients يتم طلب هذه البيانات في حال كان مصرح للعميل الاطلاع عليها.
- يتم التواصل بين كلاً من الخادم والعميل او بصيغة أخرى Back-End وبين Front-End عن طريق Http وAPI.

والآن لنستعرض بعض الصور والأشكال التي توضح ما قيل سابقاً مع حفظ جميع الحقوق لأصحابها، كالتالي:






Frontend
HTML
CSS
JS
jQuery
ANGULARJS
VS
Backend
php
JAVA
python
JS
SQL


Mobile
Storage
Applications
CLOUD
COMPUTING
Server
Database
Private
cloud
Public
cloud
Hybrid
cloud

3- Http/Https:

وهو عبارة عن بروتوكول يهتم بنقل البيانات بين الخادم والعميل وضمان وصول هذه البيانات وهو اختصار لجملة Hyper Text Transfer Protocol، ويعتمد على آلية الطلب Request من العميل والاستجابة Response من الخادم، بمعنى عندما نقوم بطلب أي موقع من الانترنت فأن المتصفح سيقوم اولاً بالبحث عن الخادم الذي يوجد به ملفات هذا الموقع وبعدها يتم انشاء اتصال Http/Https بناء على السيرفر الموجود فيه الموقع (يدعم Http او يدعم Https) ومن ثم يُرسل طلب Request إلى الخادم يطلب منه صفحات هذا الموقع وعندها يقوم الخادم اولاً بالتأكد من أن هذا العميل مصرح له للوصول إلى هذه البيانات (وفي حال كانت البيانات عامة لكل عميل يتم تجاوز هذه الخطوة) فاذا كان مصرح له يتم ارسال الملفات الضرورية إليه مثل ملفات الجافا سكريبت وملفات css وملفات HTML واي بيانات أخرى مثل ملفات Json، ...الخ، بحسب طريقة برمجة هذا الموقع والاستجابة الخاصة به، وأخيراً عند الانتهاء يتم اغلاق الاتصال بين الخادم والعميل.

وتجدر الإشارة ان كل اتصال منفصل بذاته عن أي اتصال آخر ولو كان لنفس الخادم والعميل ولنفس الموقع، بمعنى عند اجراء اتصال بين الخادم والعميل وتم خدمة العميل واغلاق الاتصال فإن الخادم لن يحفظ أي بيانات تخبره انه تم خدمة هذا العميل سابقاً، بحيث إذا قام العميل بإجراء اتصال آخر لنفس هذا الخادم ولنفس الموقع فإن السيرفر (الخادم) سوف يُعامل هذا العميل على انه عميل جديد ولا يعلم انه اجرى اتصال معه سابقاً وارسل له البيانات، ومما قيل سابقاً يتضح انه لدينا مشكلة وهي في حال قام المستخدم بتسجيل الدخول والانتقال بين أجزاء الموقع او تطبيق الويب فإنه في كل جزء يحتاج من التطبيق بيانات من السيرفر فعندها سيتم اجراء اتصال Http/Https لطلب هذه البيانات من الخادم وعندها سيطالبه الخادم بالقيام بعملية تسجيل الدخول لأنه لا علم لديه انه نفس المستخدم الذي اجرى اتصال سابقاً وتم تسجيل دخوله وهذا بطبيعة الحال مزعج ويعكس تجربة مُستخدم سيئة لتطبيق الخاص بنا، وقد تم حل هذا الامر بطرق متعددة اشهرها وأكثرها استخداماً عن طريق تقنية JWT او باستخدام Socket Api (الأخيرة طريقة برمجية معينة تحافظ على الاتصال بين الخادم والعميل دون انقطاع بحيث أي تغيير في البيانات على السيرفر يتم تحديثها بشكل مباشر عند العميل مثل قواعد بيانات Firebase).

وبطبيعة الحال يوجد بروتوكولات أخرى في عالم الانترنت غير Http/Https منها بروتوكول نقل الملفات FTP وبروتوكول TCP/IP والذي يعتبر هو أساس الانترنت ويعتمد عليه بروتوكول Http/Https.

والآن بعد إعطاء هذه المقدمة البسيطة سنقوم الآن بتوضيح الفرق بين Http vs Https.

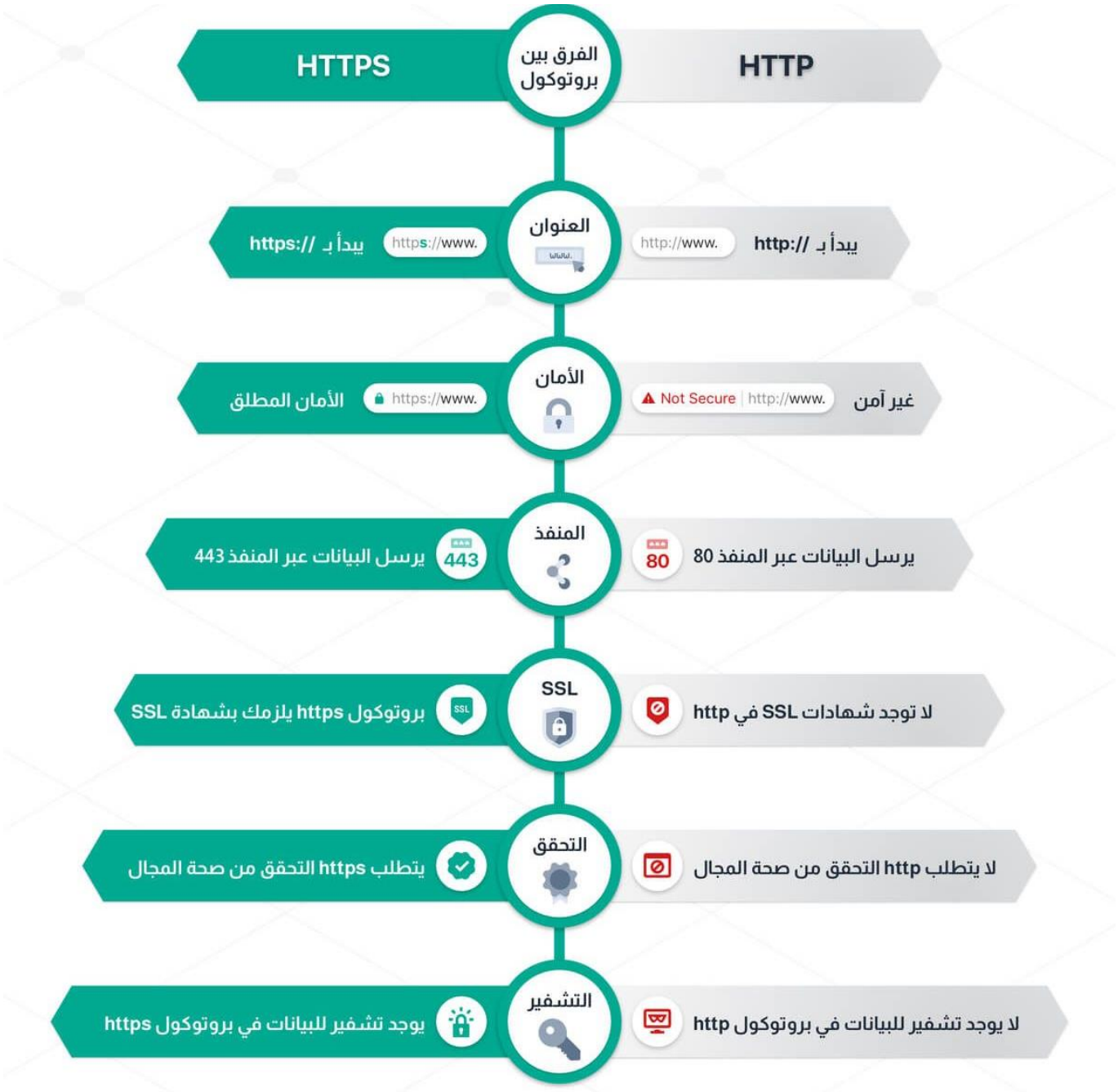
1-3- الفرق بين Http vs Https:

Https هو اختصار لجملة Hyper Text Transfer Protocol Secure أي اتصال Http الآمن، والمقصود بالآمن هو ان البيانات التي يتم نقلها بين الخادم والعميل عبره تكون مشفرة بعكس الhttp العادي، بمعنى لو كنت عزيزي المتعلم تستخدم بروتوكول http فقط وقمت بإرسال او استقبال أي بيانات بين جهازك وبين الخادم (السيرفر) فإنه من الممكن

اعتراض هذا الاتصال والاطلاع على هذه البيانات وتزيد الخطورة في حالة كانت هذه البيانات حساسة كأن تكون كلمات مرور او ارقام بطاقات فيزا، ...الخ.

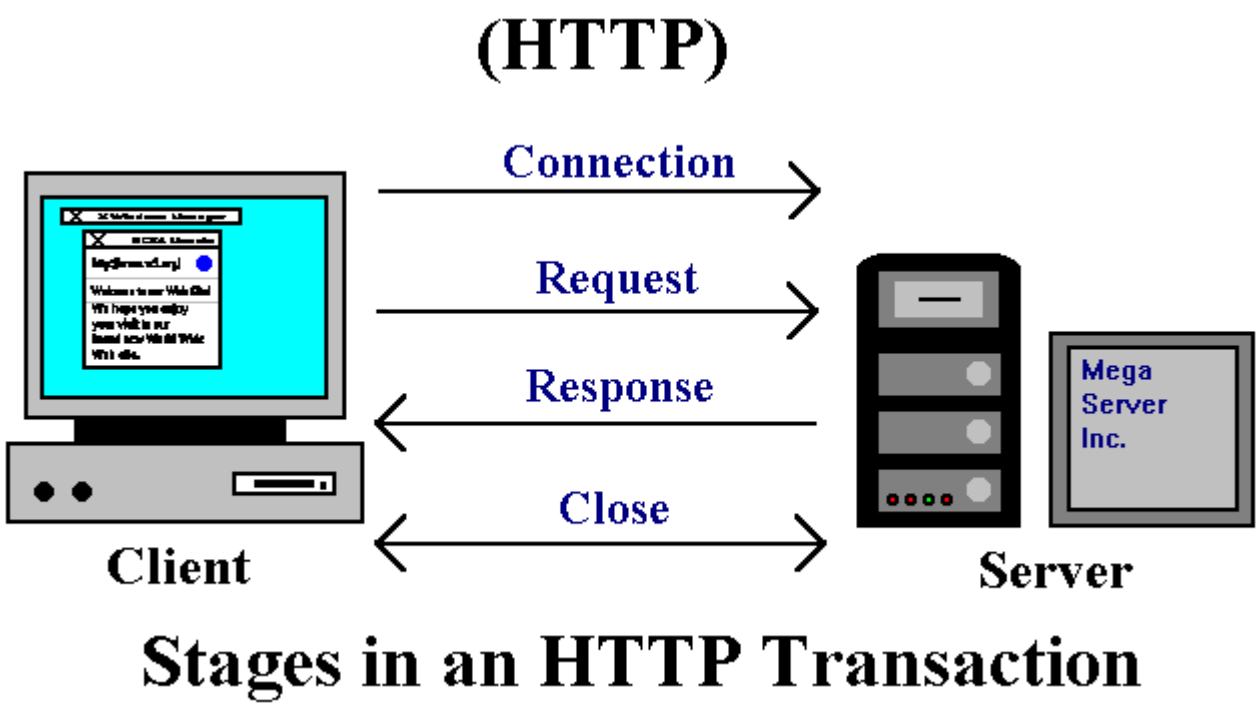
ومن اشكال التهديدات ايضاً في اتصال http العادي والتي تضر بجهاز السيرفر او بمعنى اصح بالموقع الذي يعتمد باتصال على http عادي هو امكانية مزودي خدمة الإنترنت (ISPs) أو غيرهم من الوسطاء بحقن المحتوى في صفحات الويب دون موافقة مالك الموقع. يأخذ هذا عادةً شكل الإعلان، حيث يقوم مزود خدمة الإنترنت الذي يهدف إلى زيادة الإيرادات والارباح لضخ الإعلانات المدفوعة في صفحات المواقع الخاصة بعملائه. وعند حدوث ذلك، لا يتم مشاركة أرباح الإعلانات ومراقبة جودتها بأي حال من الأحوال مع مالك الموقع الفعلي.

وفي الشكل التالي يمكن حصر الفروقات بين http وhttps، كالتالي: (جميع الحقوق محفوظة لأصحابها)

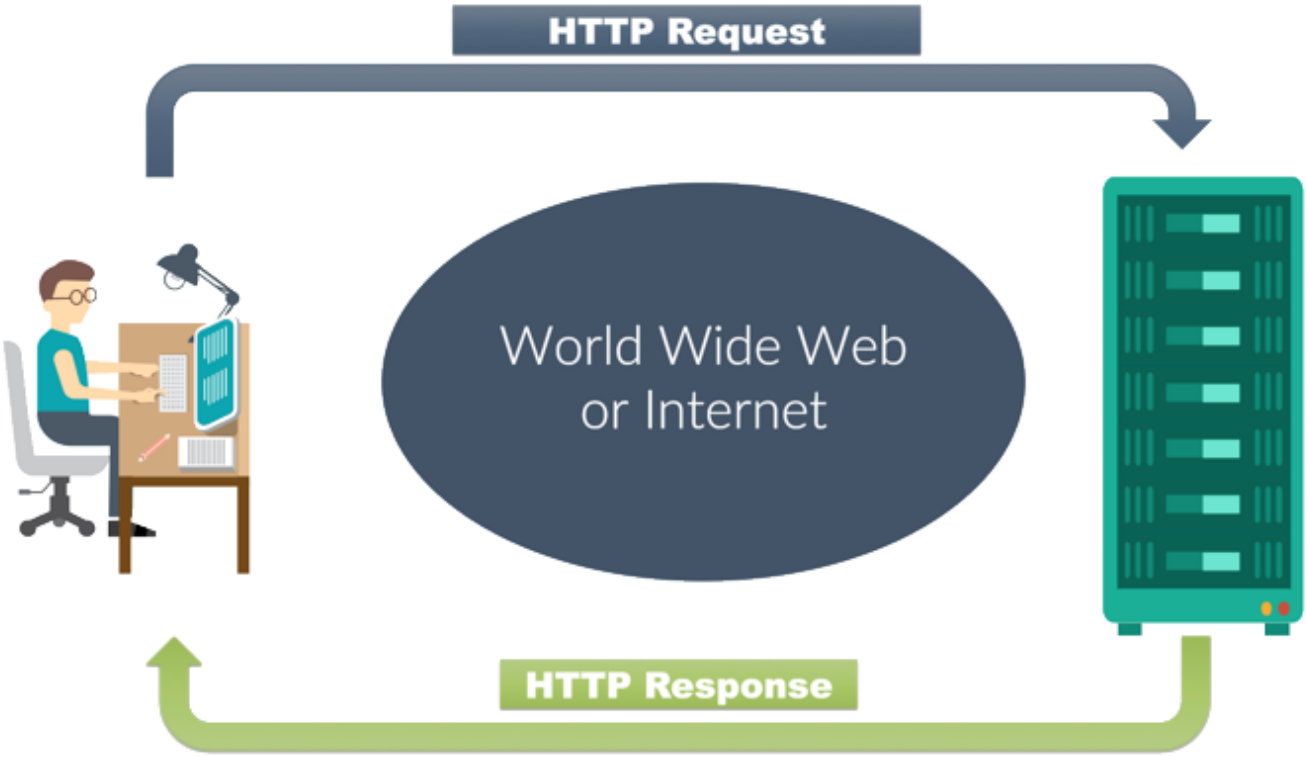


Https Request/ Https Response -2-3:

في اتصال Http لجلب وتبادل البيانات بين الخادم والعميل تتم عن طريق فكرة بسيطة وهي ارسال طلب من جهاز العميل إلى جهاز السيرفر سواء لجلب بيانات او حذفها او إضافة بيانات جديدة او التعديل عليها، ومن ثم يقوم الخادم (السيرفر) بالتأكد من نوع الطلب المرسل ومن اتاحة هذه البيانات لهذا العميل من عدمه وهل هذه البيانات هي موجوده بالأصل ام لا وهذا هذا العميل يمتلك الصلاحيات الكافية، ومن ثم يقوم بإرسال استجابة سواء كانت بإرسال هذه البيانات لجهاز العميل او عند حدوث أي خطأ يقوم بإرسال استجابة توضح لجهاز العميل ان هنالك خطأ ما سواء في الصلاحيات او البيانات غير موجوده وغيرها من الاستجابات الأخرى والتي سوف نتطرق لها لاحقاً بإذن الله، ويمكن توضيح ما قيل سابقاً من خلال الاشكال التالية: (جميع الحقوق محفوظة لأصحابها)



Stages in an HTTP Transaction



3-3- Https Status Codes:

قلنا سابقاً ان العميل Client عن طريق اتصال Https يقوم بإرسال طلب Request وبناءً على هذا الطلب يقوم الخادم (Server) بإرجاع استجابة Response، وهذه الاستجابة تكون لها رموز توضح لنا كمطوري واجهات امامية Front-End حالة هذه الاستجابة من حيث نجاحها او بعض المعلومات عن هذه الاستجابة او مثلاً هنالك أخطاء من جهة Server او هنالك أخطاء من جهة العميل وغيرها من الحالات الاخرى، لكي نتعامل معها سواء بإظهار البيانات في حال النجاح او بإعادة المحاولة او بإظهار رسالة خطأ معينة للمستخدم في حالة الفشل وغيرها من الأمور الكثيرة، والتي سنتطرق لها في هذا الجزء، وهذه الحالات يُطلق عليها Https Status Codes ويطلق عليها ايضاً Response Status Codes.

وتجدر الإشارة ان هذه الحالات والرموز الخاصة بها هي رموز مُتعارف عليها ويتم ارجاعها لنا كمطوري Front-End عن طريق مطوري Back-End، ولكن ليس اجباري على هؤلاء المطورين ان يلتزمون بهذه الرموز فمثلاً قد يرسل احد مطوري Back-End بعض الرموز العامة كأن يرسل الرمز 400 والذي يدل على أن الطلب من العميل غير صالح وهو رمز عام بينما مطور آخر يقوم بإرسال رمز أكثر تحديداً لتحديد المشكلة كأن يرسل 404 والذي يدل على ان الصفحة المراد عرضها او البيانات المراد جلبها غير موجودة.

لذلك لابد ان يكون هنالك تفاهم بين هؤلاء المطورين لكي تتضح الصورة بينهم، ولكن بشكل عام وبنسبة 99.99% يلتزم مطوري Back-End بهذه الرموز بحسب الحالات الخاصة بها.

هذه الرموز كثيرة لذلك يتم تقسيمها إلى مجموعات بحيث كل مجموعة ترمز لحالات معينة، ويتم ترقيم هذه المجموعات بقيم رقمية، كما في الجدول التالي:

رقم المجموعة	اسم المجموعة
1	Informational Responses
2	Successful Responses
3	Redirectional
4	Client Errors
5	Server errors

وكل حالة من الحالات يرمز لها برمز يتكون من ثلاث خانات، الخانة الأولى من اليسار ترمز إلى رمز المجموعة التي تنتمي إليها هذه الحالة، اما الرمزين الآخرين فهما يختصان بالحالة نفسها، فمثلاً الرمز 200 يدل على انه تم جلب البيانات بنجاح من السيرفر وجميع الأمور تسير بدون أي مشاكل، وهذا الرمز ينتمي إلى المجموعة الثانية لأنه ابتدأ بالرقم 2 والرقمين الآخرين وهما 00 فهما يختصان بالحالة نفسها ويُشيران اليها.

ويمكن استعراض جميع الحالات، مقسمة بحسب المجموعات التي تنتمي إليها مع اسم كل حالة، كما في الشكل التالي:
(جميع الحقوق محفوظة لأصحابها)

1XX Informational	
100	Continue
101	Switching Protocols
102	Processing

2XX Success	
200	OK
201	Created
202	Accepted
203	Non-authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3XX Redirectional	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
308	Permanent Redirect

4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

4XX Client Error Continued	
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
444	Connection Closed Without Response
451	Unavailable For Legal Reasons
499	Client Closed Request

5XX Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

HTTP STATUS CODES

When a browser requests a service from a web server, an error may occur.
This is a list of HTTP status messages that might be returned.

لا تقلق عزيزي المتعلم فأغلب هذه الرموز قليلة الاستخدام بالنسبة لنا كمطوري Front-End، وما نحتاجه نتعلمه ونعرفه من رموز مقارنة بإجمالي عدد الرموز يكاد يكود بسيط جداً، ويمكن حصر ما نحتاجه من الحالات ورموزها شائعة الاستخدام في عالم Front-End من خلال الجدول التالي:

1XX	Informational Responses	----------
2XX	Successful Responses	• (200) OK • (201) Created • (204) No Content
3XX	Redirectional	• (304) Not Modified
4XX	Client Errors	• (400) Bad Request • (401) Unauthorized • (403) Forbidden • (404) Not Found • (405) Method Not Allowed • (409) Conflict
5XX	Server errors	• (500) Internal Server Error

المصدر https://www.restapitutorial.com/httpstatuscodes.html

كما نلاحظ المجموعة الأولى Informational Responses لا يوجد بها حالات شائعة الاستخدام، والحالات المضمنة بها لا نحتاجها في هذا الكتاب بشكل خاص وقد لا تحتاجها انت عزيزي المتعلم في طول مسيرتك البرمجية كـ Front-End Developer.

اما باقي المجموعات فإخذنا منها الحالات شائعة الاستخدام، وهذا لا يعني ان باقي الحالات غير مهمة ولكن نأخذ ما نستخدمه بشكل متكرر واما الباقي فتستطيع الاطلاع عليه والبحث عن عمله في حال حاجتك اليه.

والآن لنقوم بإعطاء نبذة عن هذه الحالات ومتى يتم استخدامها، كالتالي:

Statuses	Description
Status Code **200** Status Name **OK**	تعتبر هذه الحالة من أشهر حالات Https، والتي تدل على ان الطلب تم بنجاح بغض النظر عن نوع هذا الطلب هل هو إضافة بيانات إلى قاعدة البيانات او حذفها او جلب هذه البيانات، وهذه الحالة لابد ان تُعيد بيانات فإذا كان الطلب جلب بيانات فيتم الرد بالبيانات التي تم طلبها، اما إذا كان حذف بيانات فيمكن مثلاً ان يتم الرد من السيرفر

Statuses	Description
Status`s Group **Successful Responses**	برسالة تبين انه تم الحذف بنجاح، اما إذا كان الطلب إضافة او تعديل فيمكن ان يقوم السيرفر بإرسال البيانات التي تم اضافتها او تعديلها، بمعنى بالمُجمل لابد ان تكون الاستجابة من السيرفر مُحملة ببيانات معينة.
Status Code **201** Status Name **Created** Status`s Group **Successful Responses**	يتم الرد من السيرفر بهذا الرمز في العادة بحالة تم إضافة بيانات جديدة لقاعدة البيانات، كإضافة مستخدم جديد او منتج معين او ...الخ، او في حالة التعديل على بيانات موجودة كأن نقوم بتغيير بيانات مستخدم معين محفوظ سابقاً في قاعدة البيانات، مع ارجاع هذه البيانات التي تم انشائها او اضافتها إلى العميل Client، ولا يتم ارسال هذه البيانات إلا بعدما يتم انشائها فعلياً في قاعدة البيانات اما في حالة تم قبول الطلب من العميل وجاري إضافة او تعديل البيانات فيتم الرد 202. مع العلم ان السيرفر ممكن ان يقوم بالرد باستخدام 200 او 201، فكما قلت سابقاً فهذه الحالات ليست الزامية ولكن هو عُرف متبع بين اغلب مطوري أنظمة السيرفرات او ما يُسمى Back-End.
Status Code **204** Status Name **No Content** Status`s Group **Successful Responses**	في العادة يتم استخدام هذه الحالة وارسال هذا الرمز من السيرفر بحالة حذف بيانات من قاعدة البيانات، ومن اهم خصائص هذه الحالة ان الرد القادم من السيرفر لا يحمل أي بيانات، لذلك نستخدم هذه الحالة في حالة نجاح الطلب وبنفس الوقت لا نحتاج أي بيانات من السيرفر.
Status Code **304** Status Name **Not Modified** Status`s Group **Redirection**	هذه الحالة تنتمي إلى حالات إعادة التوجيه ففي كل مرة نقوم بطلب ملف معين او بيانات معينة او حتى رابط معين من السيرفر ويقوم هذا السيرفر بإعادة التوجيه هذا الطلب من العميل إلى رابط آخر او مسار آخر في السيرفر فعندها يتم استخدام هذه المجموعة، ومن هذه الحالات حالة 304، ولكن هذه الحالة بالتحديد في العادة يتم استخدامها بغرض آخر مختلف، ففي بعض الحالات وخصوصاً في المواقع الكبيرة والتي تستقبل طلبات كثيرة قد تصل إلا ملايين الطلبات لنفس البيانات ففي هذه الحالة لا نُريد ان نُرهق السيرفر بجلب البيانات وانما نقوم بعمل cache او تخزين مؤقت لهذه البيانات في المتصفح ومن ثم نتأكد هل البيانات تغيرت فإذا تغيرت يتم ارسال البيانات الجديدة مع الرمز 200 اما إذا لم تتغير فيتم ارسال الرمز 304 والاكتفاء بالنسخة من البيانات الموجودة لدى العميل Client.
Status Code **400**	هذه أولى حالات مجموعة الأخطاء القادمة من العميل Client والتي يتم ارساله في الطلب إلى السيرفر ولكن السيرفر يجد خطأ في هذا الطلب Request ومن ثم يقوم بإرجاع رمز

Statuses	Description
Status Name **Bad Request** Status`s Group **Client Errors**	الحالة المناسبة للخطأ الحاصل، وأول هذه الأخطاء التي سنتطرق إليها هي حالة 400، وهذه الحالة تُشير وجود خطأ بنفس الطلب وهذا الخطأ قد يكون في صياغة البارامترات المُرسلة في رابط الAPI او هنالك خطأ في صياغة JWT Token او هنالك خطأ في Content-Type المرسلة في الHeader، او قد يكون الخطأ بأرسال طلب Request لجلب بيانات او حفظها لسيرفر تختلف عن شكل ونوع البيانات التي يتوقعها هذا السيرفر، كأن نرسل طلب لجلب البريد الالكتروني لموظف معين ولكن لا يوجد حقل في قاعدة البيانات يحتوي على نفس هذا الاسم من ضمن بيانات هذا الموظف.
Status Code **401** Status Name **Unauthorized** Status`s Group **Client Errors**	هذه الحالة مهمة ويتم استخدامها في حالة قام مستخدم معين بمحاولة الوصول إلى بيانات محمية ولم يقم بعملية تسجيل الدخول على السيرفر او حاول اجراء عملية تسجيل الدخول باسم مستخدم صحيح ولكن كلمة السر غير صحيحة، ففي هذه الحالة قد يتم الرد من السيرفر بالرمز 401.
Status Code **403** Status Name **Forbidden** Status`s Group **Client Errors**	اما هذه الحالة فتشير إلى ان السيرفر تعرف على المستخدم ولكن غير مصرح له الوصول إلى بيانات معينة في حالة حاول الوصول إليها، كأن يحاول مستخدم معين الوصول إلى API لاستعراض بيانات معينة ولكن السيرفر تحقق من صلاحيات هذا المستخدم ووجد انه لا يحق له الوصول لهذه البيانات فعندئذً قد يتم الرد من السيرفر بالرمز 403.
Status Code **404** Status Name **Not Found** Status`s Group **Client Errors**	ولعل هذه الحالة من أشهر الحالات والتي تُشير إلى ان البيانات التي تم طلبها غير متوفرة على السيرفر ومن اشكالها كأن يقوم المستخدم بمحاولة الدخول باسم مستخدم غير موجود بالأساس في قاعدة البيانات او ان يقوم مثلاً بطلب بيانات او صفحة او ملف غير موجود، ...الخ، ففي هذه الحلات السابقة وغيرها من حالات قد يتم الرد من السيرفر بالرمز 404.
Status Code **405**	

Statuses	Description
Status Name **Method Not Allowed** Status`s Group **Client Errors**	هذه الحالة في حال قام العميل Client (تطبيق Front-End) بإرسال دالة (سوف نتطرق لها لاحقاً) في الطلب Request لسيرفر ولكن هذا السيرفر (متمثل في التطبيق على السيرفر) لا يدعم هذه الدالة.
Status Code **409** Status Name **Conflict** Status`s Group **Client Errors**	هذه الحالة تنتج نتيجة وجود تعارض حاصل بين عميلين او أكثر لنفس البيانات سواء كانت بيانات محفوظة في قاعدة البيانات او ملفات يقوم العميل برفعها على السيرفر او تنزيلها من السيرفر، وهذه الحالة في العادة تنتج في وجود مجموعة من الاتصالات تتم على السيرفر من مجموعة من العملاء Clients بنفس اللحظة ولنفس البيانات مما (قد) ينتج عنه تعارض بين اتصالين او اكثر لذلك يقوم السيرفر بالرد بهذا الرمز، وايضاً ممكن ينتج في حال اتصال عميل معين (طلب Request) لسيرفر وكان السيرفر يقوم بعمل تحديثات لنفس البيانات المُراد جلبها او حفظها. ومن الأمثلة كأن يقوم مجموعة من المستخدمين بنفس اللحظة بعملية تسجيل في موقع معين واجراء عملية حفظ فلذلك قد يحدث تعارض لدى بعض المستخدمين.
Status Code **500** Status Name **Internal Server Error** Status`s Group **Server Errors**	اما هذه الحالة فتنتمي إلى آخر مجموعة وهي الأخطاء الناتجة عن السيرفر، وما يهمنا بالتحديد هو الحالة 500 حيث تشير إلى وجود خطأ داخلي في السيرفر.

4-3- Https Methods:

كما أشرنا سابقاً وبالتحديد في الحالة ذات الرمز 405 تحت اسم Method Not Allowed بان هذه الحالة تنتج نتيجة وجود دالة يطلبها العميل في الطلب Request ولكنها غير مدعومة في السيرفر لذلك تأتي الاستجابة من السيرفر بهذا الرمز.

إذن ما هي هذه الدوال؟ وما هي الفائدة منها؟

وأول خطوة للإجابة عن هذه التساؤلات هي باستعراض هذه الدوال، كالتالي: (جميع الحقوق محفوظة لإصحابها)

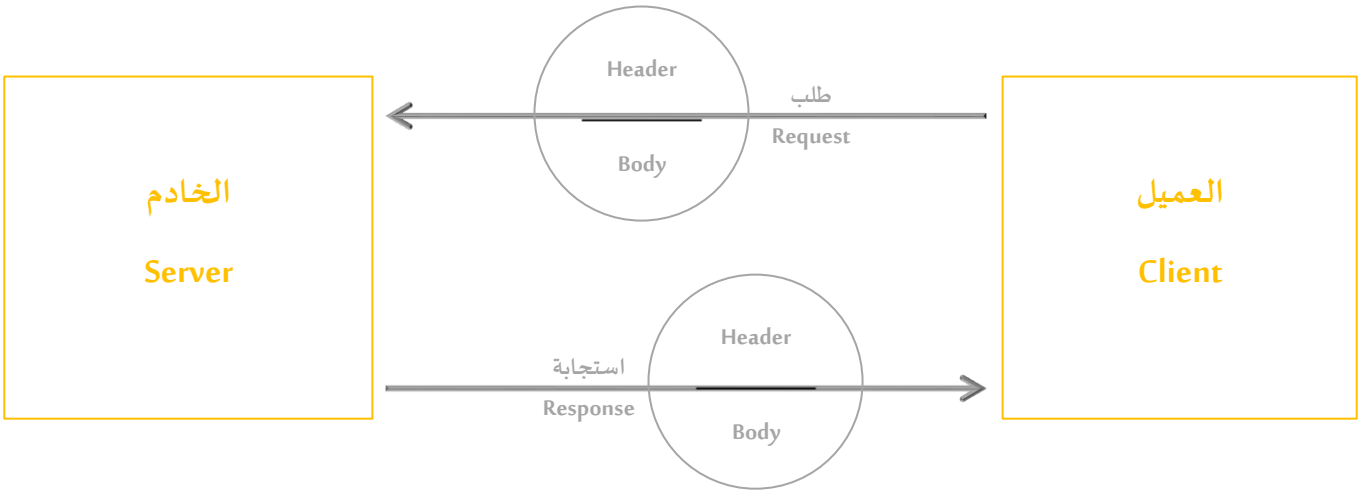
HTTP method	Description
Head	To send header part data
Trace	To resend the receive request
Put	Help us to put file into server
Delete	Remove file from server
Options	It determines which of the HTTP method server support and return appropriate error
Post	It is used to post data into server
Get	It used to get data from server

وما يهمنا كمطوري واجهات امامية Front-End هي الدوال التي تم التأشير عليها بالعلامة الحمراء، اما الفائدة من هذه الدوال بشكل عام هي انه في حال قمنا بإرسال طلب إلى السيرفر عن طريق التطبيق الخاص بنا (الذي يُطلق عليه العميل) فأننا لابد ان نظمن هذا الطلب دالة من الدوال السابقة لكي نُعرف السيرفر ما نوعية هذا الطلب هل هو لقراءة بيانات او لحفظ بيانات او تعديلها او احذفها، ويمكن تحديد وظيفة هذه الدوال كما في الجدول التالي:

Get	**تُستخدم لجلب وقراءة البيانات من السيرفر**
Post	**تُستخدم لحفظ البيانات في السيرفر**
Put	**تُستخدم لتعديل البيانات في السيرفر**
Delete	**تُستخدم لحذف البيانات من السيرفر**

3-5- Https Header/ Https Body:

والمقصود بها بُنية او هيكل الطلب Request والاستجابة Response بين كلاً من الخادم Server والعميل Client، فلكل طلب Header/Body ونفس الوضع أيضا للاستجابة فلها Header/Body (هنالك حالة واحدة فقط يكون للاستجابة Header فقط وهي عن استخدامنا لدالة Head في الطلب فعندئذ سيقوم السيرفر بالاستجابة وذلك بإرسال Header بدون Body).



بحيث يحتوي كل Header على مجموعة من الخصائص تحدد نوع وماهي كلاً من الطلب والاستجابة، فمثلاً يحتوي Header الخاص بالطلب على نوع الدالة هل هي Get او Post او ...الخ، وتحتوي ايضاً على عنوان او رابط الموقع او API المراد جلب او إضافة او حذف او تعديل البيانات منه، كما تحتوي على نوع البيانات المُراد جلبها او حفظها او حذفها او تعديلها هل هي ملفات Json او ملفات HTML او ملفات Pdf، وايضاً قد تحتوي على طريقة جلب او قراءة البيانات هل نُريد تحميل ملف Download او رفع ملف، وايضاً قد نطلب من السيرفر إعادة توجيهنا إلى رابط آخر نمرره له عن طريق هذا Header، وفي الحقيقة هنالك الكثير والكثير من الخصائص التي نستطيع ان نمررها في الطلب عن طريق Header، كما نستطيع مثلاً ان نُضيف خصائص خاصة بنا ليست موجودة بشكل افتراضي في Header كأن نُضيف JWT او ان نُضيف بارامترات معينة ، وجميع هذه الخصائص عند وصولها إلى السيرفر يقوم بقراءتها ومن ثم تنفيذ الاجراء المناسب، وعندها يقوم بالرد (الاستجابة)، وايضاً هذه الاستجابة تحتوي هي الأخرى على Header ويحتوي هو ايضاً على خصائص منها على سبيل المثال Status Code وايضاً الدالة التي قام بتنفيذها سواء كانت Get او Post او ...الخ، وتحتوي ايضاً على نوع البيانات المرسلة هل هي Json او ملف HTML او ملف وسائط او ...الخ، وغيره الكثير وايضاً كما قيل في Request Header يُقال في Response Header حيث يمكن ان يقوم مطوري Back-End بإضافة خصائص إضافية قد يحتاجها التطبيق، وعند وصول الاستجابة فإن المتصفح يقرأ الHeader الخاص به وبناءً عليها يقوم بمهام معينة.

وكما قلنا سابقاً الخصائص في الHeader كثيرة ومن الصعوبة حصرها جميعاً، ولكن قد نتطرق إلى أهمها عند وصولنا إلى التطبيق العملي بإذن الله.

وتجدر الإشارة ان الHeader يتم صياغته على شكل كائن Object جافا سكريبت، بحيث يتم كتابة الخاصية وفي مقابلها نكتب قيمة هذه الخاصية، كما في الشكل التوضيحي التالي:

```
header({
Property1: value,
Property2: value,
Property3: value,
... etc.
})
```

وسوف نتطرق بالتفصيل بإذن الله عن كيفية التعامل مع Header واضافة الخصائص له في عملية الطلب عند وصولنا إلى جزء Angular HttpClient.

اما Body فهو البيانات نفسها التي يتم تبادلها بين الخادم والعميل، سواء كانت بصيغة Json او ملفات HTML، ...الخ.

مع العلم انه هنالك بعض الحالات التي قد يتم ارسال واستقبال البيانات بين الخادم والعميل عن طريق الHeader، وهذا يرجع لكيفية بناء التطبيق على الخادم.

4- JavaScript Object Notation (JSON):

هذه التقنية هي عبارة عن صيغة ثابته ومتعارف عليها لمشاركة البيانات بين جميع لغات البرمجة، والملف المُحتوي على هذه البيانات ينتهي بالامتداد (json.اسم الملف)، وطريقة صياغة البيانات بداخلها مشابهة لكائنات الجافا سكريبت، لذلك يُفترض بنا كمطوري Front-End ان نكون ملمين بهذه التقنية وعلى علم بهيكلتها وطريقة صياغة البيانات فيها لأنه كما أشرت قبل قليل ان JSON يعتبر كنسخة مشابهة لكائنات الجافا سكريبت، ولكن لا يمنع ان نمر عليها بشكل سريع لزيادة التوضيح، حيث تتكون من جزئيين جزء يُسمى key والجزء الثاني هو القيمة لهذا الkey وهذه القيمة قد تكون أي قيمة سواء مصفوفة او نص او رقم او تاريخ او حتى ممكن ان تكون كائن فرعي آخر، كما في الشكل التالي:

```
{
  "firstName": "Foo",
  "lastName": "Bar",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "Cairo"
  },
  "phoneNumbers": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "fax", "number": "646 555-4567" }
  ]
}
```

فكما نلاحظ في المثال السابق لدينا بيانات على شكل صيغة JSON حيث تتكون من key وقيمة value فعلى سبيل المثال firstName هو عبارة عن key والقيمة الخاصة به هي "Foo" بينما address هو عبارة عن key وقيمته عبارة عن كائن

فرعي يحتوي هو الآخر على اثنين Keys والقيم الخاصة بهما، بينما الkey ذو الاسم phoneNumbers قيمته عبارة عن مصفوفة من الكائنات، وهكذا نستطيع تمثيل أي بيانات نريدها على شكل JSON.

ولكن السؤال المهم هو لماذا JSON؟

اولاً سهلة الفهم وسهلة القراءة من قبل المبرمجين والتعامل معها ابسط من غيرها كـ XML، وايضاً تتعرف عليها أي لغة برمجة، وسهلة التحويل إلى كائن جافا سكربت والعكس صحيح.

لذلك تم اعتمادها بشكل كبير كطريقة لتخاطب وارسال البيانات بين الخادم والعميل، حيث يقوم التطبيق الموجود على الخادم بعد جلب البيانات من قاعدة البيانات تحويلها إلى صيغة JSON وارسالها في Body مع وضع القيمة application/json الخاصية content-type في headers، لكي يفهم العميل (المتصفح) ان هذه البيانات بصيغة JSON، وايضاً عند الطلب Request من العميل للخادم، يتم تحديد نوع البيانات application/json (Angular وبشكل افتراضي يتعامل مع أي طلب على انه JSON مالم نُغير نوع البيانات في Headers) وعند استقبالها في الخادم يتم قراءتها وفهمها ومن ثم اتخاذ الاجراء المناسب بحسب الدالة المُرسلة في الطلب.

5- Application Programing Interface (API):

مفهوم API ليس بالجديد فهذا المفهوم مستخدم وبكثرة لدى مبرمجي التطبيقات بكافة اشكالها، فمثلاً عند قيامنا ببرمجة تطبيق سطح المكتب ليعمل على نظام تشغيل ويندوز فإننا قد نحتاج إلى بعض ملفات النظام (ملفات dll في الغالب) والتي تحتوي بداخلها على مجموعة من الدوال البرمجية المبرمجة مسبقاً من قبل مبرمجي نظام تشغيل ويندوز لكي نستفيد منها في تطبيقنا للقيام بمهام معينة، وهذه الدوال تُسمى API، بحيث نحن كمبرمجي تطبيقات لا يهمنا كيف تم بناء هذه الدوال وانما يهمنا بالدرجة الأساسية كيف نستفيد منها لتنفيذ المهام التي نُريدها، وقس على ذلك عزيزي المتعلم جميع ما يتم ما تستدعيه داخل تطبيقك سواء كانت مكتبات خارجية او ملفات نظام او حتى كلاسات قمت انت ببنائها مسبقاً لكي تستفيد منها في تطبيقات أخرى بدلاً من ان تقوم ببناء هذه المهام في كل مرة تقوم بها ببرمجة تطبيق معين يقوم بنفس هذه المهام.

فجميع هذه الشفرات البرمجية الخارجية تُسمى API، ومع الوقت انتقل هذا المفهوم لعالم الويب وبالتحديد في عملية تبادل البيانات بين الخادم والعميل، وبتحديد أكثر في Back-End.

وقبل التطرق لشرح مفهوم واستخدامات API في عالم الويب نحتاج في البداية لتوضيح كيف يتم سابقاً تبادل البيانات بين الخادم والعميل.

قبل دخول مفهوم API في عالم الخادم والعميل والويب بشكل عام، كان يتم الطلب من العميل للخادم ومن ثم يقوم الخادم بأغلب العمل حيث يحلل هذا الطلب ومن ثم يتصل بقاعدة البيانات اذا كانت هذه الصفحة تتطلب بيانات موجودة في قاعدة البيانات ومن ثم يقوم بتضمينها بملف HTML وملفات الجافا سكريبت وملفات التنسيقات css، واخيراً

يقوم بالاستجابة وهي ارسال هذه البيانات إلى العميل، وعند وصولها إلى العميل يقرئها المتصفح ويقوم بتحميلها لعرضها للمستخدم، وعند أي طلب يقوم به العميل وتأتي الاستجابة فلابد ان يقوم المتصفح بإعادة تحميل الصفحة.

والطريقة السابقة يعيبها بعض الأمور، ممكن تلخيصها بالنقاط التالية:

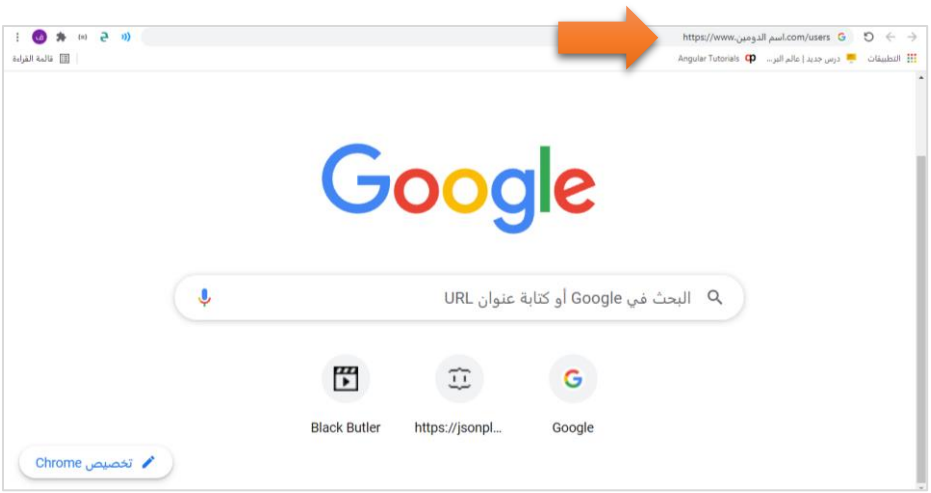
- عند كل اتصال من العميل للخادم لجلب بيانات لابد من إعادة تحميل الصفحة، ولو كان الطلب لجلب بيانات جزء من الصفحة وليس الصفحة كاملة، وهذا يؤدي إلى استهلاك موارد الخادم وتأخير جلبها وعرضها في المتصفح.
- لو افترضنا ان هنالك نظام كبير ويتم استخدامه عن طريق اكثر من عميل، عميل يمثل موقع ويب وعميل آخر يمثله تطبيق موبايل وعميل آخر يمثله تطبيق سطح المكتب، فجميع هذه التطبيقات تتصل بنفس قاعدة البيانات، ولكن بهذه الطريقة لابد ان نعمل تطبيق Back-End لكل تطبيق من هذه التطبيقات سواء تطبيق ويب او تطبيق موبايل او تطبيق سطح المكتب، وهذا لا يخفي عليك عزيزي المتعلم كم الميزانية التي سوف يتم رصدها لتنفيذ مثل هذه الأنظمة.
- الاتصال المباشر بقاعدة البيانات بين Front-End وBack-End، ومما قد ينتج بعض المشاكل الأمنية.
- وما يعبها ايضاً هو دمج Front-End (ملفات HTML وملفات JavaScript وملفات css) مع Back-End، مما يجعل هنالك صعوبة في تطوير تطبيقات Front-End بمعزل عن تطبيقات Back-End.

إذن بعد استعراض جميع هذه العيوب، يتراود إلى الاذهان تساؤل مهم وهو، ما هو الحل؟

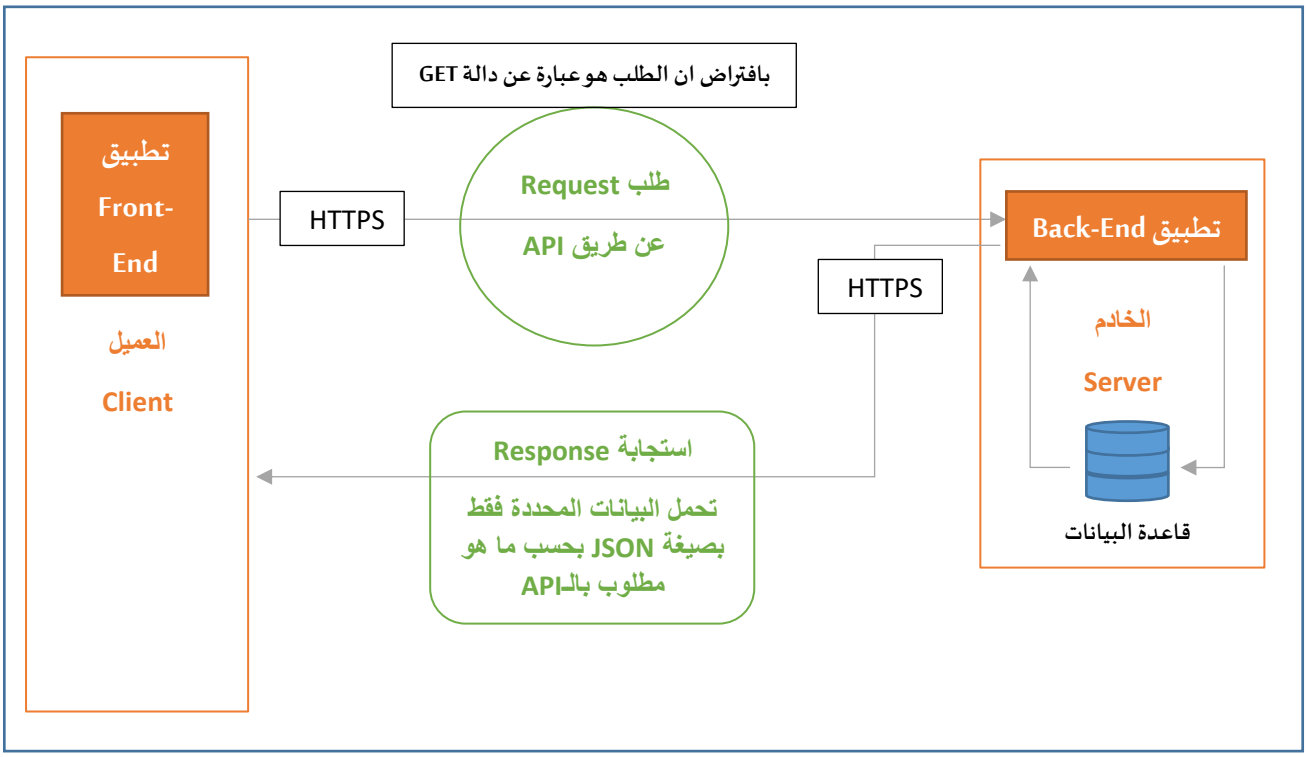
والحل يأتي بمفهوم API حيث يتم انشائها من قبل مطوري Back-End وتكون على شكل رابط URL حيث كل رابط يشير إلى جزء معين من البيانات، فمثلاً الرابط التالي:

https://www.اسم الدومين.com/users

نفترض ان هذا الرابط سوف يقوم بجلب بيانات المستخدمين، وتتم هذه العملية بالشكل التالي:



وعندها سيتم ارسال طلب Request من العميل (المتصفح) إلى الخادم الموجود على الدومين المُعطى في الرابط، وعند وصول الطلب سيقوم تطبيق Back-End الموجود على الخادم بالاتصال بقاعدة البيانات لجلب البيانات المحددة ومن ثم يتم ارسال الاستجابة وهي عبارة عن هذه البيانات بصيغة JSON، كما في الشكل التوضيحي التالي:



بعد وصول البيانات إلى العميل يتم عرضها للمستخدم، وبذلك قمنا بحل اغلب إشكاليات الطريقة القديمة، كالتالي:

- نستطيع جلب بيانات محددة وعرضها فقط بدون الحاجة إلى إعادة تحميل الصفحة جميعها مع كل طلب.
- تطبيق Back-End واحد فقط يخدم جميع أنواع التطبيقات فمثلاً لو كان العميل هو تطبيق موبايل فيتم خدمته وارسال البيانات له عن طريق هذا الرابط وبنفس الطريقة لجميع الأنواع الأخرى فكل الذي نُريده كمبرمجي Front-End هو الرابط الذي عن طريقه نستطيع جلب البيانات.
- لا يتم الاتصال مباشرة بقاعدة البيانات وانما تتم عن طريق API ومن ثم يتم الاتصال بقاعدة البيانات.
- هنالك فصل تام بين Back-End وFront-End فكل الذي يربط بينهم هو روابط API وهذا يتيح لنا تطوير Front-End بمعزل عن Back-End، وليس هذا فحسب بل انه يدخل بصلب مفهوم API الذي اشرنا له في بداية كلامنا، حيث كل الذي نهتم به كمطوري واجهات امامية Front-End هو **كيفية الاستفادة من هذه API لجلب او إضافة او حذف او تعديل البيانات بغض النظر عن كيفية بناء هذه الدوال وجميع العمليات التي تحدث في الخلفية.**

وفي الحقيقة ان روابط API ليس فقط رابط ثابت لجلب بيانات معينة وانما ديناميكي وذلك عن طريق مثلاً ارسال اسم المستخدم عن طريق هذا الرابط لجلب بيانات مستخدم معين، وايضاً نفس الرابط نقوم كمطوري واجهات امامية بإرسال اسم مستخدم آخر عن طريق نفس الرابط لجلب بيانات مستخدم آخر وهكذا في كل مرة نستطيع تحميل هذا الرابط ببيانات مختلفة، كما اننا نستطيع تحميل الرابط بباراميترات كأن نجري عملية بحث وفق معايير معينة كما فعلنا

في مثال البحث في تطبيق الأفلام في الفصل الثاني من هذا الكتاب، ومن هذا المنطلق نستطيع ان نقول ان جميع تعاملاتنا بين الخادم والعميل تتم عن طريق روابط API سواء للبحث او تسجيل مستخدم جديد او لإجراء عملية تسجيل دخول او خروج او ...الخ.

ولا تقلق عزيزي المتعلم من ناحية الأمان فنستطيع ان نحدد اسم دومين محدد للوصول إلى هذا الرابط، بحيث لو وقع لدى اشخاص آخرين فلا يستطيعون الوصول إلى هذه البيانات، إلا في حالة كانت هذه API هي عامة.

وبما أننا تطرقنا إلا API العامة، فنستطيع ان نقول أن مفهوم API أدى إلى نقله نوعية كبيرة فمثلاً نشاهد كثيراً تطبيقات أخرى تستطيع الوصول إلى محتويات تطبيق twitter وعرض التغريدات وهذا معناه انه يوجد رابط او مجموعة روابط API قامت بوضعها شركة twitter للوصول إلى أجزاء معينة من البيانات، وقس على ذلك اغلب تطبيقات التواصل الاجتماعي، وليس هذا فحسب الم تتساءل يوماً عزيزي المتعلم كيف عند الرغبة بتسديد مخالفاتك المرورية فإنك تتوجه إلا أي جهاز صراف الكتروني للبنك الذي تتعامل معه او بشكل اسهل تفتح التطبيق الالكتروني الخاص بهذا البنك وبمجرد وضعك لرقم هويتك الوطنية او رقم الإقامة الخاصة بك ستظهر جميع مخالفاتك مع ان مخالفاتك مفترض ان تكون من ضمن بياناتك في خوادم وزارة الداخلية، اذن بكل تأكيد ان مطوري Back-End لخوادم وزارة الداخلية اتاحة API معينة تسمح للبنوك بالاطلاع فقط على المخالفات التابعة لهذا الشخص مع إمكانية سدادها، وفي الحقيقة الأمثلة كثيرة ولا يسعنا ذكرها.

إذن ما هي أنواع API؟

قام الباحثون بتقسيمها إلى مجموعة من الأنواع، كالتالي:

- عامة: وهي متاحة للجميع ومجانية بحيث تستطيع الوصول لها وهي في الغالب يتم استخدامها لتجربة فقط كموقع https://jsonplaceholder.typicode.com/
- مشروطة: وهي مجانية ولكن تتطلب key معين يتم تقديمه من قبل Back-End لكي نستطيع الوصول إلى هذه API، كمثلاً مواقع التواصل الاجتماعي.
- مدفوعة: وهي API تقدم بيانات ذات قيمة بحيث لابد ان يتم الدفع لشركة او الجهة المسئولة عنها لكي نصل إلى هذه البيانات وعرضها في التطبيق الخاص بنا.
- خاصة: وهي API خاصة بنظام معين ويتم استخدامها فقط من قبل عملاء هذا النظام، كأن يكون لدينا نظام خاص بشركة معينة ولا يتم استخدامه إلا من قبل موظفي هذه الشركة فقط.

6- JSON Web Token (JWT):

قلنا سابقاً ان عند انتهاء الاستجابة وارسالها إلى العميل فإن الاتصال ينقطع بين الخادم والعميل، وعند اجراء نفس العميل لاتصال آخر فإن الخادم لا علم لديه بالاتصال الأول ويعامله كأنه اول اتصال، وهذا يولد لدينا مشكلة وهي عند اجراء عملية تسجيل الدخول على النظام فإن العميل سيقوم بإجراء اتصال HTTPS وارسال طلب إلى الخادم يحمل

جميع بيانات اسم المستخدم وكلمة المرور عن طريق API معين، وعند وصول الطلب الى الخادم سيتخذ بعض الإجراءات لتأكد من ان البيانات صحيحة وعندها سيرسل استجابة بصحة البيانات، إلى هنا الأمور جيدة ولكن بعد عملية تسجيل الدخول يفترض من تطبيق Front-End ان يقوم بإرسال طلب آخر لجلب بيانات كالصفحة الشخصية لهذا المستخدم الذي قام بتسجيل الدخول، عندها تحدث المشكلة التي تحدثنا عنها سابقاً وهي ان الخادم لا علم لديه بالاتصال السابق مع هذا العميل وسيعامله كأنه اتصال جديد، لذلك سيرفض جلب البيانات لهذا العميل لأنه لم يتعرف عليه، إذن ما هو الحل وفي الحقيقة هناك مجموعة من الحلول اشهرها وأكثرها شيوعاً هي استخدام تقنية JWT وهي عبارة عن رمز طويل يتم توليده من قبل الخادم عند اجراء عملية تسجيل الدخول ويتم تخزينه في الخادم وايضاً يتم ارساله إلى العميل في Header، ويقوم العميل متمثلاً في التطبيق بتخزين هذا JWT في أي مكان بالتطبيق – بالغالب في Local Storage – وعند أي اتصال من العميل بالخادم يتم ارسال هذا JWT في Header وعند وصوله إلى الخادم يقوم الخادم باتخاذ بعض الإجراءات لتأكد من ان هذا JWT ينتمي إلى هذا العميل ونفس المخزن لديه وايضاً لا يزال صالح – لان له تاريخ انتهاء – وعندها سيتعرف الخادم على هذا العميل.

هذا باختصار مفهوم هذه التقنية ونحن كمطوري واجهات امامية لا نهتم بطريقة توليد هذا JWT لأن هذا من عمل مطوري Back-End فكل الذي نهتم اليه هو التأكد من وجوده في الاستجابة وتخزينه ومن ثم ارساله في الطلب.

الفصل الثاني

المفاهيم الأساسية

في مكتبة Rxjs

1- مقدمة:

لو قمنا بتوسيع أفاق تفكرينا قليلاً من ناحية جميع تعاملنا مع الحاسب الآلي لوجدنا أن كل ما نقوم به هو تعامُل مع البيانات بشكل مباشر او غير مباشر من حيث تخزين البيانات او ارسال واستقبال البيانات او تحرير وتعديل هذه البيانات، هذا من جهة المستخدمين والذي ينعكس بطبيعة الحال على التطبيقات التي تقوم بتنفيذ جميع العمليات على البيانات، ومن هذا المنطلق لابد ان يمتلك مطورو التطبيقات المهارات الضرورية لتعامل مع البيانات في تطبيقاتهم، فلابد ان يُجيد أي مطور تطبيقات – بجميع أنواعها المختلفة – كيفية مرور هذه البيانات في تطبيقه وكيف يتم قراءة وعرض هذه البيانات في التطبيق وفي حال تم استقبال بيانات من خارج التطبيق – قاعدة بيانات – وخصوصاً إذا كانت بيانات متفرقة ويتم عرضها في جزء واحد كيف يتم تجميعها وعرضها في هذا الجزء المحدد من التطبيق او من ناحية هنالك نوع من البيانات موجودة بأكثر من مكان في التطبيق وفي حال التعديل عليها يتم التعديل في جميع الأماكن الموجودة فيها هذه البيانات، وحقيقة أوجه الأمثلة كثيرة وليس هنا المكان لعرضها جميعاً.

ومما قيل سابقاً نحتاج نحن كمطوري تطبيقات إلى مكتبات او تقنيات تسهل وتساعدنا في التعامل مع البيانات وإدارتها في تطبيقاتنا، ومن هذه التقنيات تقنية ReactiveX التي تحتوي على مجموعة من المفاهيم والتقنيات والدوال التي تساعد المطورين على إدارة تدفق البيانات في تطبيقاتهم.

حيث تم تطوير تقنية ReactiveX في بداية الأمر من قِبل شركة مايكروسوفت لإطار عمل .NET. لتعامل مع البيانات الغير التزامنية Asynchronous Data، ومن ثم تم التوسع في استخدامها لتشمل كثير من لغات البرمجة وأطر العمل المختلفة، وتستطيع عزيزي المتعلم الذهاب إلى الموقع الرئيسي لهذه التقنية باسم ReactiveX.io ومن ثم اختيار صفحة languages او عن طريق هذا الرابط (http://reactivex.io/languages.html)، وسوف تجد جميع لغات البرمجة التي تدعمها هذه التقنية، وما يهمنا منها مكتبة باسم Rxjs حيث انها مخصصة للغة الجافا سكربت وأطر العمل الخاصة بها كالAngular وهي اختصار لجملة Reactive Extensions for JavaScript، حيث تحتوي على عدد من التقنيات والدوال التي تساعدنا للتعامل مع البيانات الغير تزامنية.

2- البيانات التزامنية والغير تزامنية Asynchronous vs Synchronous:

وبعض الأحيان يتم تسميتها بالكود المتزامن والغير متزامن، والمقصود بالبيانات التزامنية هو انه في حال التعامل معها فإن التطبيق سوف يبقى في حالة انتظار وجمود إلى ان يتم جلب هذه البيانات وعرضها للمستخدم، ولها استخداماتها ولعل من أشهر استخداماتها هو عند اجراء عملية تسجيل الدخول فإن التطبيق سوف يبقى في حالة انتظار ولن يقوم بأي عملية إلى ان يصل الرد من السيرفر، وفي حال أردت تغيير هذه الحالة بحيث ان التطبيق الخاص بك يستطيع تنفيذ أي كود آخر او تنفيذ مهام أخرى سواء بطلب المستخدم او بشكل تلقائي **اثناء فترة انتظار جلب هذه البيانات** ففي هذه الحالة تصبح البيانات غير تزامنية، ولها استخدامات كثيرة جداً منها على سبيل المثال في حال كان لدينا نموذج تسجيل مستخدم جديد ونريد من المستخدم ان يتحقق من اتاحة البريد الإلكتروني، ففي هذه الحالة نُرسل طلب لسيرفر لتأكد من ان هذا البريد الإلكتروني متاح من عدمه، وبنفس الوقت لا نجعل المستخدم ينتظر الرد وانما نُتيح لها استكمال

وتعبئة بيانات النموذج وممكن ايضاً ان نُرسل طلب آخر لسيرفر كأن نظهر له المدن التي تختص دولة معينة ...الخ، ومتى تم الرد من السيرفر يتم التعامل معها وإظهارا للمستخدم، ففي هذه الحالة تسمى هذه البيانات بالبيانات الغير تزامنية أي اننا ممكن ان نُرسل اكثر من طلب لسيرفر دون الانتظار لمعرفة الرد وانما نجعل التطبيق يقوم بهام أخرى وعند رجوع أي رد من السيرفر نتعامل معه.

وممكن حصر أشهر استخدامات البيانات الغير تزامنية في JavaScript – ليس الكل – من خلال النقاط التالية:

- دالة ajax في JavaScript.
- دالة setTimeout وأخواتها في JavaScript.
- الاتصال وجلب البيانات من قواعد البيانات.
- القراءة والكتابة على الملفات.

وهذا النوع من البيانات مهم جداً ويجب على كل مطور تطبيقات إجادة التعامل معها، وفي الحقيقة هنالك عدة تقنيات تُساعد مطوري لغة JavaScript لتعامل مع هذا النوع من البيانات منها (Rxjs - CallBack - Promises - Async/Await).

3- لماذا مكتبة Rxjs؟

أشرت سابقاً ان هنالك تقنيات متعددة لتعامل مع البيانات الغير تزامنية Async Data، ولكن السؤال الذي يطرح نفسه لماذا مكتبة Rxjs عوضاً عن بقية التقنيات الأخرى، وقبل الإجابة عن هذا السؤال يجب الإشارة انني لن اتطرق لشرح مفاهيم التقنيات الأخرى وايضاً افترض بإن القارئ لديه معرفة سابقه بهذه التقنيات وفي حال لم يكن هنالك أي معرفة مسبقة الرجاء الرجوع الى العديد من المصادر التي تتكلم عن Callbacks, Promises and Async/Await.

اما للإجابة عن السؤال الرئيسي، لماذا مكتبة Rxjs؟ فنستطيع الإجابة عن هذا السؤال من خلال النقاط التالية:

- Rxjs تتعامل مع تدفق من البيانات Streams Data وليس مع قيمة واحدة فقط، كما في التقنيات الأخرى.
- Rxjs تحتوي على كم هائل من الدوال التي تتيح التعامل مع البيانات.
- Rxjs تتميز بأنه يمكن الغاء Stream of Data وإعادة الاتصال بآخر، كأن يكون لدينا قائمة وعندما يقوم المستخدم بالضغط على عنصر من القائمة يتم الاتصال بقاعدة بيانات لجلب البيانات التي تخص هذا العنصر، ويُحتمل ان يقوم المستخدم بالضغط على عنصر ومن ثم يقوم مباشرة بالضغط على عنصر آخر قبل اكتمال جلب بيانات العنصر الأول، ومن هذا المنطلق يأتي دور هذه الميزة حيث يتم الغاء الطلب او Stream Data الخاص بالعنصر الأول وإعادة الاتصال لجلب البيانات الخاصة العنصر الثاني.
- Rxjs تتميز بانها تستطيع التعامل مع مصادر مختلفة من البيانات مثل قاعدة البيانات او الملفات او User Events (سواء من لوحة المفاتيح او الفأرة) وغيره من المصادر باستخدام تقنيات واحدة، وليس لكل مصدر من مصادر هذه البيانات تقنياته الخاصة به.

- Rxjs تتميز بانها تتيح دمج مجموعة مختلفة من البيانات في Stream واحد وعرضها في مكان واحد في التطبيق، كأن نقوم بجلب بيانات مستخدم معين في stream وصورة العرض الخاصة به في stream آخر ومن ثم دمجها باستخدام دوال معينة تُقدها هذه المكتبة في stream واحد ومن ثم عرض هذه البيانات في مكان محدد في التطبيق.
- Rxjs تتميز بانها تراقب أي Stream من البيانات ومن ثم في حال تغير القيمة تقوم بجلبها بشكل تلقائي.
- وأخيراً Rxjs تُقدم مجموعة متنوعة من التقنيات لمعالجة الإخطاء التي قد تحدث في البيانات.

4- العمليات الرئيسية الثلاثة في تقنيات ReactiveX:

في تقنيات Reactive Programming او اختصارً ReactiveX هنالك ثلاثة مصطلحات (عمليات) رئيسية يجب فهمها والإلمام بها جيدً، وهي:

- Observable
- Observer
- Subscription

وسوف اتطرق إلى هذه المصطلحات من خلال مكتبة Rxjs بالتحديد، بما يضمن فهمك لها بدون الدخول بالتفاصيل التي قد تشتت انتباهك.

4-1- Observable:

أشرنا سابقاً أن جميع تعاملنا مع الحاسب يتعلق بالبيانات سواء بشكل مباشر أو غير مباشر، وفي الحقيقة Observables هي البيانات التي نُريد أن نتعامل معها من خلال التطبيق الخاص بنا، لذلك نستطيع ان نقول ان Observable هو عبارة عن تدفق من البيانات بجميع أشكالها (أرقام – حروف – مصفوفات – كائنات – أحداث Events – استجابة من السيرفر من خلال HTTP – ...الخ) تمر من خلال التطبيق الخاص بنا ونستطيع ان نخضعها لمجموعة هائلة من العمليات بما ويتناسب مع احتياجنا، بحيث تقدم مكتبة Rxjs كمية هائلة من العمليات – سوف أُفرد لها باب كامل – تساعدنا بالتلاعب مع هذه البيانات بما نحتاجه، فمثلاً لو افترضنا انه تم جلب مجموعة من البيانات من قاعدة البيانات تخص مستخدم معين على هيئة Json وبنفس الوقت تم جلب مجموعة من البيانات الأخرى والتي ايضاً لها علاقة بهذا المستخدم، ففي هذه الحالة وغيرها نستطيع الاستفادة من هذا العمليات Operators كأن نأخذ بعض البيانات فقط ونهمل الباقي وبنفس الوقت البيانات التي تم أخذها نستطيع ندمجها في مصفوفة او كائن واحد، ونُعالج الإخطاء ان وجدت، بما يضمن ظهور هذه البيانات في التطبيق بدون أي مشاكل.

ولنرجع الآن إلى Observable حيث كما قلنا ان هذه البيانات هي التي نُشير إليها بهذا المصطلح، وهنالك مسميات أخرى لها منها Stream او Observable Sequence. وايضاً هذا Stream من البيانات ممكن ان يكون له نهاية او لا يكون كأن

نعمل Observable يقوم كل ثانية بعث قيمة عشوائية معينة ففي هذه الحالة نطلق عليه Infinite Observable، وأخيراً مما يميز Observable انها قد تكون بيانات غير تزامنية او تزامنية.

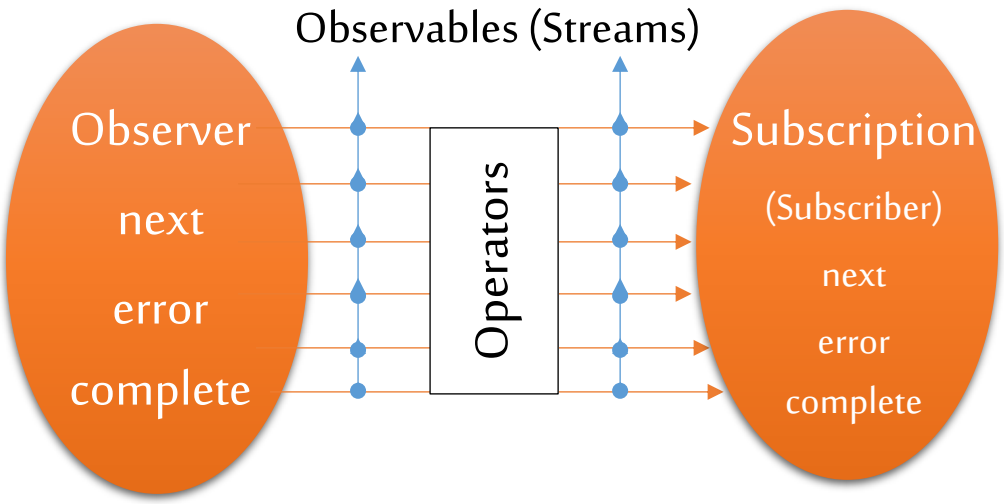
2-4- Observer:

قلنا ان Observable هي البيانات التي تتدفق عبر التطبيق، ولكن إلا نحتاج إلى مصدر يقوم بعمل بعث Emit لهذه البيانات. وهذا المصدر يُطلق عليه Observer، وهو عبارة عن كائن يحتوي على ثلاث دوال:

- next(): ويتم تشغيل هذه الدالة في حال الحصول على قيمة جديدة (بيانات جديدة).
- error(): ويتم تشغيلها في حالة وجود أخطاء في البيانات.
- complete(): ويتم تشغيلها في حالة اكتمال Observable، ويجب الانتباه انه في حالة Infinite Observable فإن هذه الدالة لن يتم تشغيلها، لذلك يجب الانتباه في حالة كتابة أي اسطر برمجية في هذه الدالة.

3-4- Subscription:

أصبح لدينا Observable او Stream من البيانات وبنفس الوقت لدينا المصدر Observer الذي يعمل Emit لهذه البيانات ويتحكم بها، ولكن هنالك آخر جزء وهو متى نُريد أن يبدأ هذا Stream، وهذا ما نطلق عليه Subscription، حيث انه لابد ان نعمل Subscribe لأي Observable لكي يتم تدفق البيانات، وهو عبارة عن كائن Object **نمرر له ما يسمى Subscriber والذي يحتوي هو الآخر على الثلاثة دوال الموجودة في Observer، بحيث إذا قام Observer بعمل emit عن طريق الدالة next يتم استقبالها في الدالة التي تحمل نفس الاسم في Subscriber ونفس الطريقة مع error وcomplete**، ونستطيع تمثيل جميع هذه العمليات من خلال الشكل التوضيحي التالي:



وفي حقيقة الأمر هذا الشكل لا يُعبر بشكل دقيق ولكن كبداية ولتوضيح المفاهيم الأساسية، أستعنت به.

5- Create Observable:

جميع ما قيل سابقاً هي شرح لمفاهيم وأسس نظرية، ولم نتطرق إلى الجزء العملي، وهو الذي سوف نقوم به في هذا الجزء حيث سوف نقوم بإذن الله بتطبيق جميع المفاهيم السابقة بشكل عملي من خلال مشروع Angular جديد، ويجب ملاحظة ان مكتبة rxjs مضمنة بشكل افتراضي مع إطار عمل Angular، لذلك قم عزيزي المتعلم بإنشاء مشروع Angular جديد وليكن اسمه rxjs-demo، بحيث يكون ملف app.component.ts، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {

}
```

هنالك طرق عديدة لإنشاء Observable او ما يسمى Stream، منها عن طريق عمل instance من الكلاس Observable، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable();
}
```

نلاحظ عملنا instance من الكلاس Observable في كائن اسميته obs$ (لك حرية اختيار الاسم الذي تُريده)، وهذا الكلاس هو generic لذلك يُفضل ان نمرر نوع البيانات في هذا observable، وهنا في مثالنا هذا نوع البيانات هي نص string، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private obs$ = new Observable<string>()
}
```

ونأتي إلى اهم نقطة، وهي ما سوف نقوم بتمريره إلى هذا الكلاس، وهو عبارة عن دالة function وهذه الدالة هي Observer الذي يقوم بعمل emit للبيانات لهذا Observable عن طريق الدالة next، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable<string>(function (observer) {
    observer.next('value one');
    observer.next('value two');
    observer.next('value three');
  });
}
```

ونستطيع استخدام Arrow Function الجديدة في ES6، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private obs$ = new Observable<string>((observer) => {
      observer.next('value one');
      observer.next('value two');
      observer.next('value three');
    });
}
```

نلاحظ اننا قمنا بتمرير دالة وهذه الدالة تستقبل بارامتر واحد اسميته observer (لك حرية استخدام الاسم الذي تُريده) وهذه الدالة تمثل Observer الذي أشرنا له سابقاً حيث يقوم بإرسال او عمل emit للقيم على شكل Observable او Stream من البيانات.

والخطوة التالية هي Subscription أي لابد ان نعمل subscribe لهذا Observable لكي نستطيع ان نستقبل البيانات منه، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private obs$ = new Observable<string>((observer) => {
      observer.next('value one');
      observer.next('value two');
      observer.next('value three');
    });

  constructor() {
    this.obs$.subscribe();
  }
}
```

نلاحظ اننا قمنا بعمل subscribe للمتغير obs$، والآن نستطيع ان نمرر Subscriber لهذا Subscription على شكل كائن Object له ثلاث خصائص مشابهه لدوال الموجودة في Observer وهي next – error – complete، وكل خاصية تستقبل دالة، وهذه الدالة تحمل القيم المرسلة من Observer، فمثلاً الخاصية next تستقبل القيم المرسلة من Observer عن طريق الدالة next والخاصية error تستقبل القيم المرسلة من Observer من خلال الدالة error، ونفس الأمر بالنسبة للخاصية complete، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
```

```
  private obs$ = new Observable<string>((observer) => {
      observer.next('value one');
      observer.next('value two');
      observer.next('value three');
    });

  constructor() {
    this.obs$.subscribe({
      next: function(value) {
        console.log(value);
      },
      error: function(err) {
        console.log(err);
      },
      complete: function() {
        console.log('completed!!');
      }
    });
  }

}
```

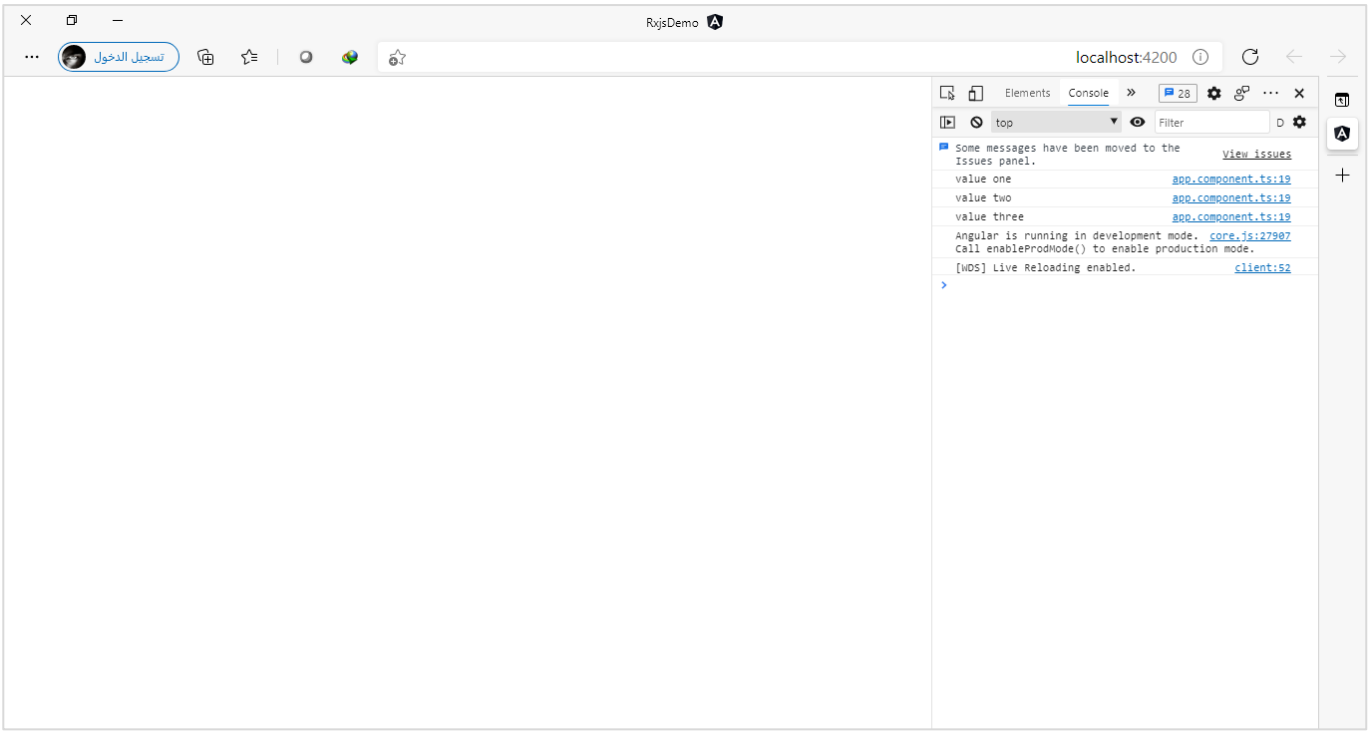
نلاحظ قمنا بتمرير subscriber على شكل كائن وهذا الكائن يحتوي على ثلاث خصائص الأولى باسم next (لابد ان تكتب بنفس الاسم) والقيمة لها عبارة عن دالة تستقبل بارامتر واحد اسميته value (لك حرية اختيار الاسم الذي تُريده) وهذا البارامتر هو الذي يحمل القيمة المرسلة من Observer عن طريق الدالة next، ونفس الوضع مع error اما الخاصية الأخيرة complete فلا تستقبل أي بارامتر لأن Observer عندما يعمل emit عن طريق الدالة complete فإنه لن يمرر أي قيمة لذلك استقبلناه هنا في Subscriber بدون أي بارامتر، ونستطيع إعادة كتابة هذا Logic البرمجي باستخدام Arrow Function بدلاً من الطريقة التقليدية لكتابة الدوال لتصبح كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable<string>((observer) => {
    observer.next('value one');
    observer.next('value two');
    observer.next('value three');
  });

  constructor() {
    this.obs$.subscribe({
```

```
      next: (value) => {
        console.log(value);
      },
      error: (err) => {
        console.log(err);
      },
      complete: () => {
        console.log('completed!!');
      },
    });
  }
}
```

اما الآن لنشاهد النتيجة في المتصفح، ولا تنسى عزيزي المتعلم ان تعمل serve للمشروع عن طريق كتابة الامر ng s -o، كالتالي:



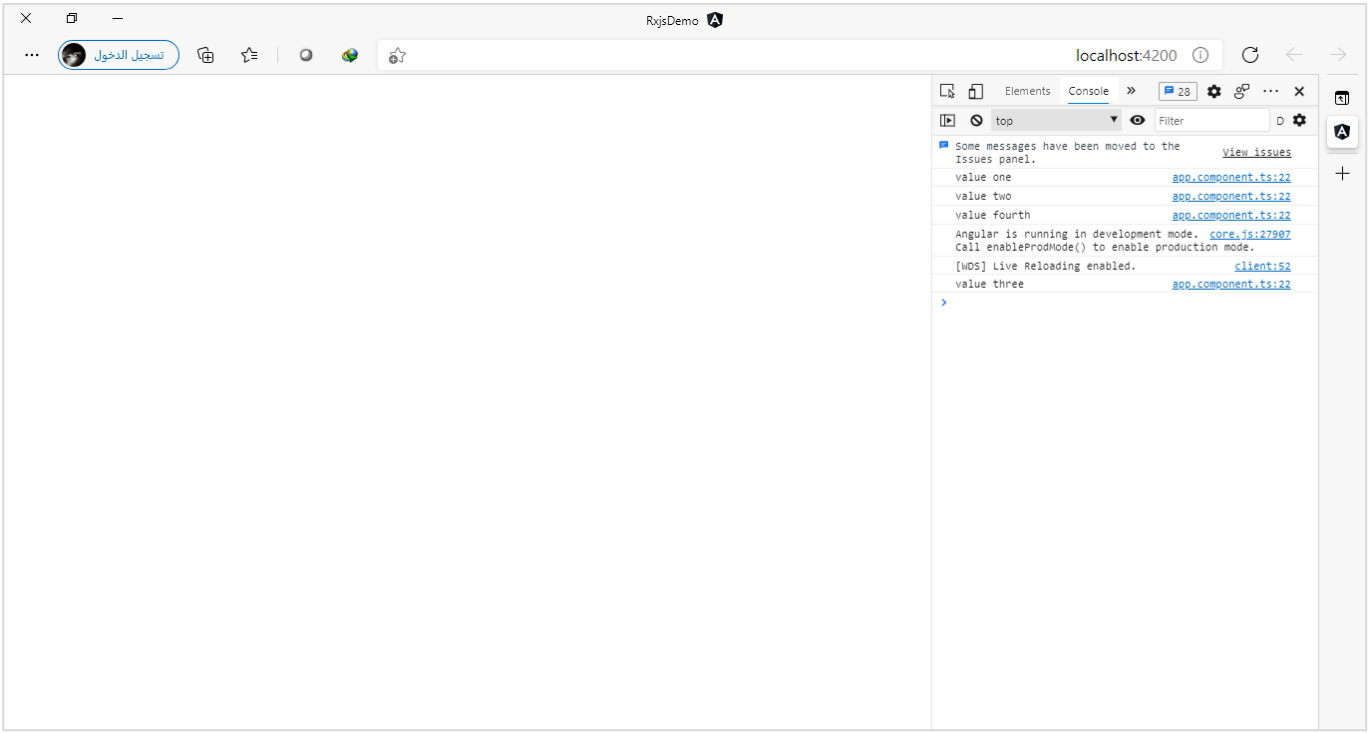
نلاحظ تم عرض القيم في console بالترتيب، ونلاحظ ان هذه البيانات هي تزامنية أي بمعنى تم عرض القيمة الأولى ومن ثم الثانية ومن ثم الثالثة، ولعمل محاكاة سوف نستخدم الدالة setTimeout لكي نحاكي بيانات غير تزامنية، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent {
  private obs$ = new Observable<string>((observer) => {
    observer.next('value one');
    observer.next('value two');
    setTimeout(() => {
      observer.next('value three');
    }, 1000);
    observer.next('value fourth');
  });

  constructor() {
    this.obs$.subscribe({
      next: (value) => {
        console.log(value);
      },
      error: (err) => {
        console.log(err);
      },
      complete: () => {
        console.log('completed!!');
      },
    });
  }
}
```

نلاحظ قمنا بتأخير عمل emit للقيمة الثالثة بعد ثانية، ولنرى النتيجة في المتصفح:



نلاحظ قام Observer بعمل emit للقيمة الأولى ومن ثم الثانية وعند وصوله إلى الثالثة وجد انه لابد ان ينتظر ثانية لكي يعمل emit لهذه القيمة لذلك لم ينتظر وانما قام بعمل emit للقيمة الرابعة، ومتى ما تم وصول القيمة الثالثة يعمل لها

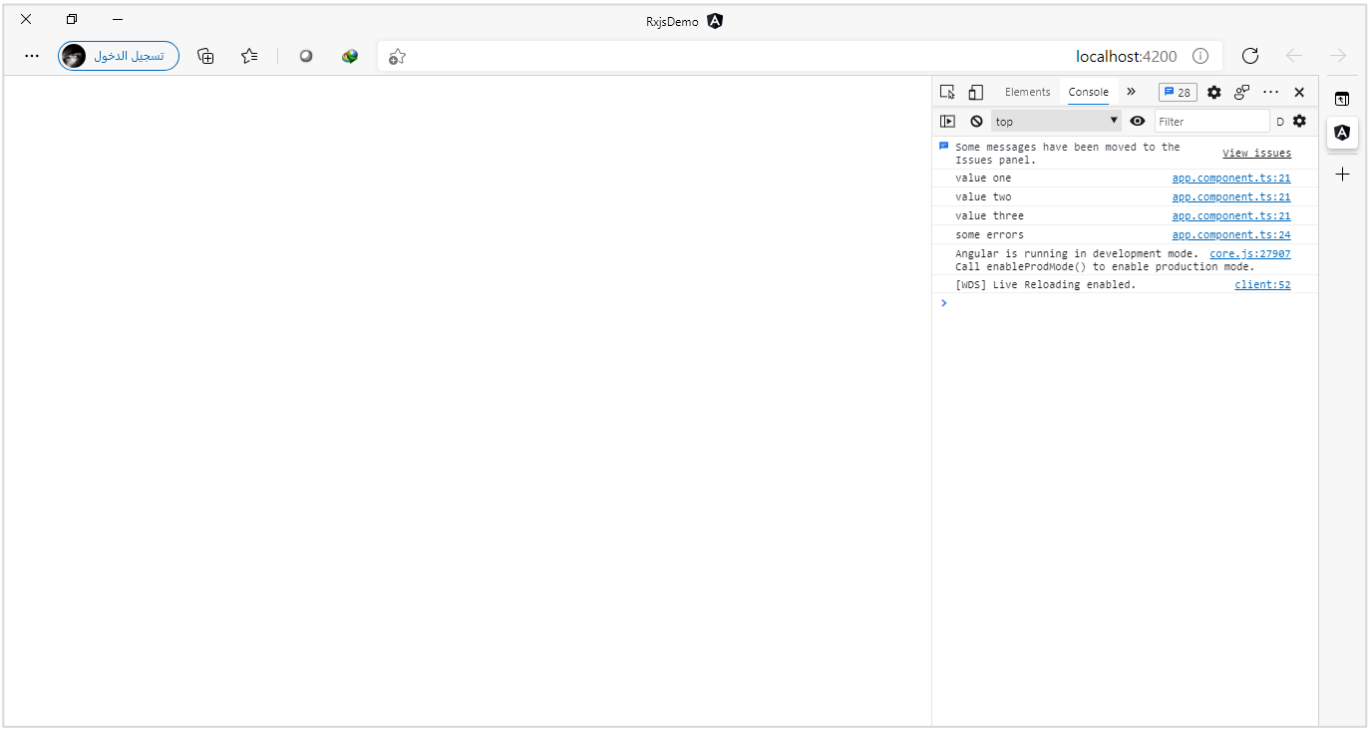
emit، وهذا هو الذي اشرت له سابقاً وهو من نقاط القوة في Observable، بحيث لا يجعل التطبيق ينتظر القيم وانما أي قيمة تصله يعمل لها emit مباشرة، مع العلم ان هنالك طرق متعددة نستطيع عن طريقها ان نجعل التطبيق ينتظر أي قيمة يعتمد عليها القيم التي تليها، ولكن نؤجل الحديث في هذه النقطة لاحقاً في هذا الكتاب بإذن الله.

لنرجع الآن إلى Subscriber وسوف نلاحظ ان دالتي error وcomplete لم يتم تشغيلها، والسبب في ذلك بكل بساطة ان Observer لم يعمل emit لأي خطأ عن طريق الدالة error او الدالة complete، ولنقوم بإرسال خطأ معين عن طريق Observer، ولنرى النتيجة:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable<string>((observer) => {
    observer.next('value one');
    observer.next('value two');
    observer.next('value three');

    observer.error('some errors');

    observer.next('value fourth');
  });

  constructor() {
    this.obs$.subscribe({
      next: (value) => {
        console.log(value);
      },
      error: (err) => {
        console.log(err);
      },
      complete: () => {
        console.log('completed!!');
      },
    });
  }
}
```

ولنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ تم عمل emit لثلاث قيم الأولى ولكن عندما قام Observer بعمل emit لخطأ معين عن طريق الدالة error قام Observable بتجاهل أي قيمة بعدها، وهذا هو المتوقع لأن أي خطأ يحدث في Stream من البيانات سوف يتم تجاهل أي بيانات قد تتدفق عبر Observable بعد هذا الخطأ، وبنفس الوقت يقوم Subscriber بتشغيل الخاصية error واستقبال القيمة من خلال البارامتر الذي تم تمريره لهذا الدالة والذي هو في حالتنا هذه اسميته err.

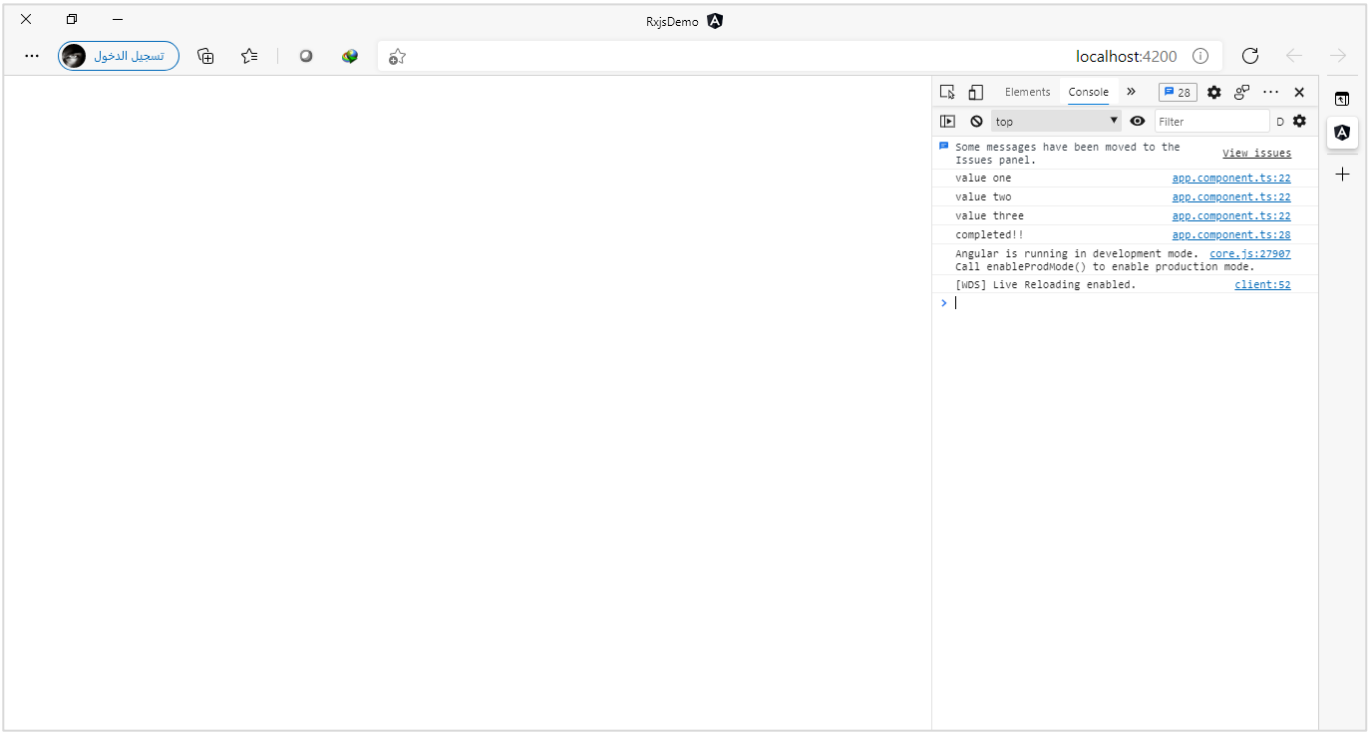
والآن لنقوم بنفس الأمر ولكن مع complete، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable<string>((observer) => {
    observer.next('value one');
    observer.next('value two');
    observer.next('value three');
    // observer.error('some errors');
    observer.complete();
    observer.next('value fourth');
  });

  constructor() {
    this.obs$.subscribe({
      next: (value) => {
```

```
        console.log(value);
      },
      error: (err) => {
        console.log(err);
      },
      complete: () => {
        console.log('completed!!');
      },
    });
  }

}
```

نلاحظ قمنا بتعطيل السطر البرمجي الخاص بإرسال الخطأ واضفنا سطر برمجي آخر يقوم فيه Observer بعمل emit لكي يُخبر Subscriber بإن Observable مُكتمل، وعندها لن يقوم باستقبال أي قيمة أخرى، ولنرى النتيجة في المتصفح، كالتالي:



نفس ما قيل في error يُقال هنا من حيث تجاهله لأي قيمة بعد complete.

ويجدر الإشارة ان Subscriber الذي تم تمريره لSubscription يمكن صياغته بعدة طرق، كالتالي:

الطريقة الأولى

```
    this.obs$.subscribe({
      next: (value) => {
        console.log(value);
      },
      error: (err) => {
        console.log(err);
      },
```

```
    complete: () => {
      console.log('completed!!');
    },
  });
```

الطريقة الثانية

```
  this.obs$.subscribe({
    next(value) {
      console.log(value);
    },
    error(err) {
      console.log(err);
    },
    complete() {
      console.log('completed!!');
    },
  });
```

الطريقة الثالثة

```
  this.obs$.subscribe(
    (value) => {
      console.log(value);
    },
    (err) => {
      console.log(err);
    },
    () => {
      console.log('completed!!');
    }
  );
```

وفي حال الرغبة بتجاهل error وcomplete يُفضل استخدام آخر طريقة this.obs$.subscribe((value) => {})

ملاحظة مهمة: اعلم عزيزي المتعلم انه في البرامج الو اقعية في الغالب لا يتم إنشاء Observable بهذه الطريقة، وقد شرحتها لكي تستطيع فهم واستيعاب المفاهيم الأساسية، مع العلم ان المفاهيم هي نفسها وفهمك واستيعابك لما قيل سابقاً بمثابة استيعابك لمكتبة rxjs بنسبة ٥٠% إلى ٦٠%، اما في البرامج الو اقعية وخصوصاً في عالم Angular يأتينا Observable جاهز ولن نقوم بعمل emit بأنفسنا عن طريق Observer و انما Angular يقوم بهذا الأمر نيابة عنا وكل الذي نهتم فيه هو جُزئية Subscription، وسوف نتكلم عن هذا الامر في قسم خاص في هذا الباب، وفي حال استدعى الأمر ان نقوم بإنشاء Observable فنقوم بهذا الأمر عن طريق دوال معينة Operators، وسوف نُفرد باب كامل بإذن الله نتكلم فيه بالتفصيل عن هذه الدوال.

6- متى يتم إيقاف Observable؟

تكلمنا سابقاً ان هنالك حالتين يتم فيها إيقاف Observable الأول عن طريق استدعاء الدالة complete بواسطة Observer والثاني في حالة وجود أخطاء، ونضيف هنا حالتين أخرى، هما ان يقوم المبرمج يدوياً بعمل Unsubscribe او باستخدام بعض Operators، وسوف نؤجل الكلام عن الحالة الأخيرة إلى ان نصل إلى الجزء الخاص Operators.

ونستطيع تمثيل الحالات الأربعة من خلال الشكل التالي:

اما الآن لنعطي مثال على حالة Unsubscribe، كالتالي:

ملف app.component.ts

```
import { Component, OnDestroy } from '@angular/core';
import { Observable, Subscription } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnDestroy {

  private subscription: Subscription;

  private obs$ = new Observable<string>((observer) => {
    observer.next('value one');
    observer.next('value two');
    observer.next('value three');
    // observer.error('some errors');
    observer.complete();
    observer.next('value fourth');
  });
```

```
  constructor() {
    this.subscription = this.obs$.subscribe({
      next: (value) => {
        console.log(value);
      },
      error: (err) => {
        console.log(err);
      },
      complete: () => {
        console.log('completed!!');
      },
    });
  }

  ngOnDestroy(): void {
    this.subscription.unsubscribe();
  }

}
```

نلاحظ قمنا بتعريف متغير من النوع Subscription وقيمته عبارة عن Subscription للObservable المتمثل في المتغير obs$، وأخيرا في دالة ngOnDestroy قمنا بعمل Unsubscribe، وهذا الامر مهم جداً حيث يرفع من اداء التطبيق لديك، ويجب على كل مبرمج معرفة هل هذا Observable يحتاج ان يكون متدفق طول فترة تشغيل التطبيق وانما يحتاجه فقط في جزء معين من التطبيق، كأن يتصل بهذا Observable لجلب البيانات بكل مرة يقوم المستخدم بالتوجه إلى الجزء الخاص بعرض بيانات المستخدمين وعندما يقوم المستخدم بالانتقال إلى جزء آخر من هذا التطبيق فلا نحتاج لهذه البيانات لذلك نقوم بعمل إيقاف لهذا Observable لكيلا نستهلك موارد الجهاز لدى المستخدمين وهذا ينعكس بشكل سيء على التطبيق لديك وانطباع المستخدمين اتجاهه.

7- Higher Order Observable:

وهي عبارة عن مجموعة من Observables المتداخلة، او بصياغة أخرى نستطيع ان نقول انها عبارة عن Observer يقوم بعمل emit لObservable آخر او مجموعة من Observables الأخرى وبنفس الوقت قد يقوم الObserver الخاص بهذه Observables الفرعية بعمل هو الآخر emit لObservables فرعية أخرى وهكذا، بحيث يصبح لدينا مجموعة Observables المتداخلة مع بعضها البعض.

ولها أمثلة كثيرة منها على سبيل المثال في حالة كان لدينا Observable يحتوي على بيانات الأقسام وبنفس هذا Observable هنالك Observable فرعي يحتوي على بيانات الموظفين، لذلك نأخذ من Observable الأول بيانات القسم المحدد ومن ثم نمرر هذه البيانات إلى Observable الفرعي الثاني لكي نستعرض فقط بيانات الموظفين التابعين لهذا القسم، اما من ناحية التطبيق العملي لنقوم بإنشاء Observable داخل Observable فرعي، كالتالي:

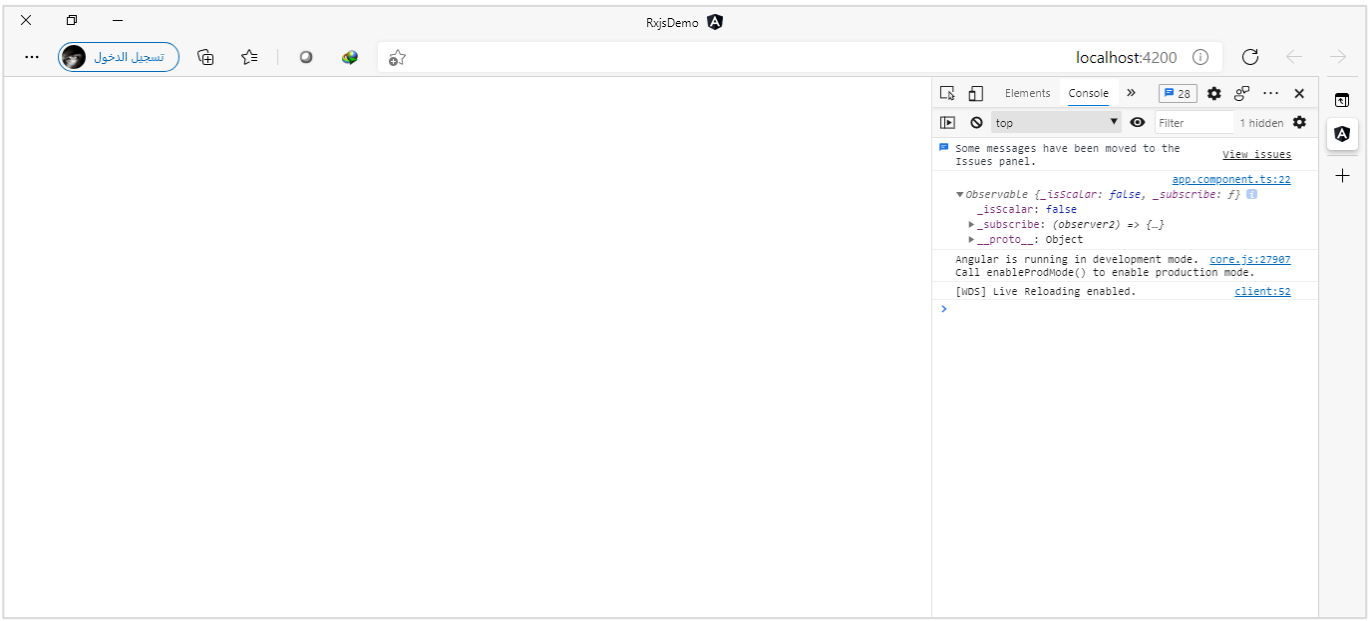
ملف app.component.ts

```
import { Component } from '@angular/core';
```

```
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable<Observable<string>>((observer) => {
    observer.next(
      new Observable<string>((observer2) => {
        observer2.next('value one from nested observable');
      })
    );
  });

  constructor() {
    this.obs$.subscribe((value) => {
       console.log(value);
    });
  }
}
```

نلاحظ يوجد لدينا Observable الأول باسم obs$ ويقوم Observer الخاص بعمل emit لـObservable آخر فرعي بحيث يقوم Observer الخاص بهذا Observable الفرعي بعمل emit لقيمة نصية.

ومن ثم قمنا بعمل Subscription لكي نستطيع قراءة القيمة النصية، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:

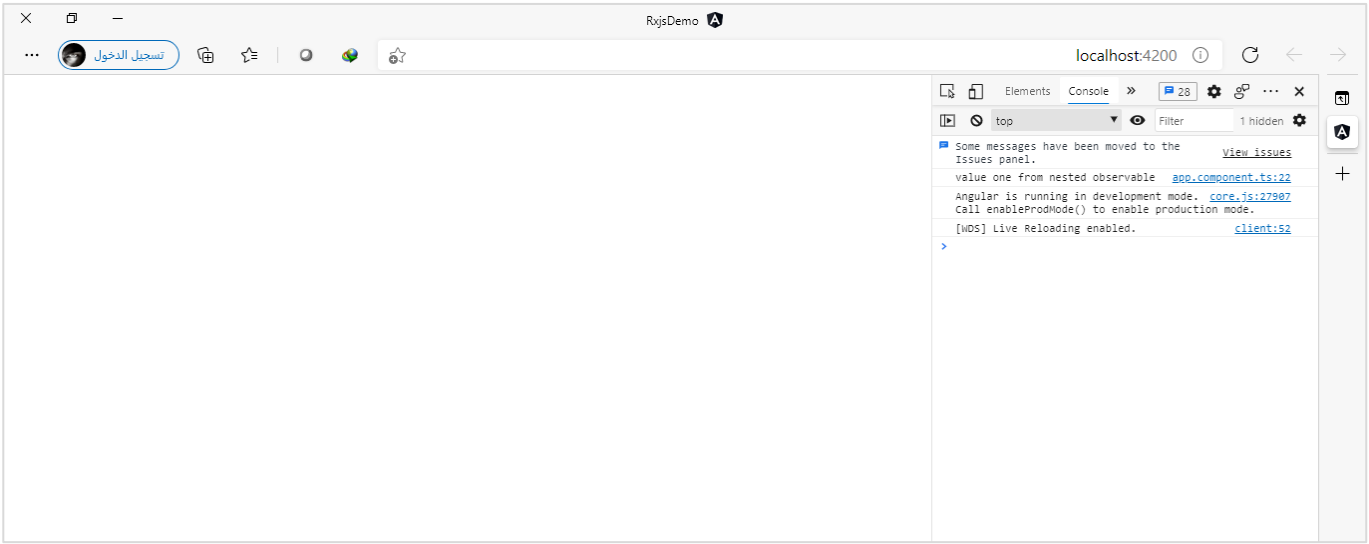


نلاحظ نتيجة Subscription للObservable الأول هو Observable الفرعي، وهذا هو المتوقع، ولكن نحن هنا لا نريد Observable نفسه وانما نريد البيانات والقيم في هذا Observable، لذلك لابد ان نعمل ايضاً Subscription لهذا Observable الفرعي لكي نستطيع الحصول على القيم، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs$ = new Observable<Observable<string>>((observer) => {
    observer.next(
      new Observable<string>((observer2) => {
        observer2.next('value one from nested observable');
      })
    );
  });

  constructor() {
    this.obs$.subscribe((nestedObs) => {
      nestedObs.subscribe((value: string) => {
        console.log(value);
      });
    });
  }
}
```

اما النتيجة في المتصفح، فتكون كالتالي:



نلاحظ استطعنا الوصول إلى القيمة في Observable الفرعي، وهكذا بقية Observables الفرعية.

ملاحظة مهمة: اعلم عزيزي المتعلم ان هذه الطريقة غير محببة ولا يُفضل استخدامها، بحيث يكون لدينا مجموعة من Subscriptions المتداخلة لكي نصل إلى قيمة معينة، و انما يُفضل استخدام طرق أخرى تقدمها لنا مكتبة Rxjs وهي عبارة عن مجموعة من الدوال Operators الجاهزة التي تم تصميمها لتعامل مع هذه الحالات، ولكن سوف نؤجل الكلام بهذا النقطة إلى ان نصل إلى الباب المخصص بشرح Operators بإذن الله تعالى.

8- Subjects:

وهو عبارة عن Observable مع إضافة بعض المميزات الأخرى، وتتكون من مجموعة من الدوال هي:

(Subject – AsyncSubject – BehaviorSubject – ReplaySubject)

اما المميزات التي تميز جميع هذه الدوال عن Observable العادي، ممكن تلخيصها من خلال النقاط التالية:

- نستطيع عن طريق Subjects ان نعمل emit للبيانات من خارج Observable بعكس الطريقة التقليدية، بحيث كنا نُعرف Observable ومن ثم نمرر له دالة والتي هي عبارة عن Observer الذي نقوم عن طريقة بعمل emit لأي قيمة، بمعنى لا نستطيع عمل التالي:

App.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private observable = new Observable((observer) => {
    observer.next('value one');
  });

  constructor() {
    this.observer.next('value one');
    this.observable.next('value one');
  }
}
```

بالطبع نستطيع ان نعمل خدعة معينة لكي نستطيع ان نعمل emit من خارج هذا Observable، كالتالي:

```
import { Component } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private obs = new Subscriber(); //OR private obs: PartialObserver<string>;
  private observable = new Observable((observer) => {
    this.obs = observer;
    observer.next('value one');
  });

  constructor() {
    this.observable.subscribe((data) => {
      console.log(data);
    });
    this.obs.next('value two');
  }
}
```

ولكن هذه الطريقة غير محببة وقد تحدث لك بعض المشاكل وتصل إلى نتائج غير متوقعة في تطبيقك.

والحل هو باستخدام Subjects حيث نستطيع بكل سهولة ان نعمل emit من خارج هذا Observable، كالتالي:

App.component.ts

```
import { Component } from '@angular/core';
import { Observable, Subject, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private subject = new Subject<any>();

  constructor() {
    this.subject.next('value one');
    this.subject.next('value two');
    this.subject.next('value three');
  }
}
```

نلاحظ قمنا بتعريف كائن من نوع الكلاس Subject وبما انه Generic قمنا بتمرير نوع لهذا Subject، وأخيراً قمنا بعمل emit لثلاث قيم من خارج هذا Subject.

- مما يميز Subjects ايضاً انها Observable وObserver وSubscriber، فهي عبارة عن عبارة عن Stream من البيانات (Observable) وبنفس الوقت نستطيع استخدامها لكي نعمل emit لأي بيانات أخرى (Observer) وأخيراً نستطيع ان نعمل Subscription لهذه البيانات لكي نستطيع قراءتها (Subscriber)، كالتالي:

App.component.ts

```
import { Component } from '@angular/core';
```

```
import { Observable, Subject, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private subject = new Subject<string>();

  constructor() {
    this.subject.subscribe((data) => {
      console.log(data);
    });

    this.subject.next('value one');
    this.subject.next('value two');
    this.subject.next('value three');
  }
}
```

والنتيجة في المتصفح، كالتالي:

نلاحظ تم عمل emit للقيم ومن ثم عملنا subscribe لنفس المتغير وبالأخير تم عرض هذه القيم في المتصفح.

- وهنالك ميزة مهمة وهي عمل مشاركة لنفس البيانات مع جميع Subscribers لنفس Observable من النوع Subjects، بعكس Observable التقليدي الذي ممكن ان يُعطي قيم مختلفة لكل Subscription يتم عليه بحيث يُعطي لك Subscriber قيمة مختلفة (ولو كانت نفس القيمة ولكن يتم ارسالها مرتين)، ولتوضيح لنعطي مثال على Observable مع مجموعة من Subscribers ومثال آخر لSubjects مع ايضاً مجموعة من Subscribers، كالتالي:

App.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
```

```
  observable = new Observable<number>((observer) => {
    observer.next(Math.random());
  });

  constructor() {
    // subscription 1
    this.observable.subscribe((data) => {
      console.log(data);
    });

    // subscription 2
    this.observable.subscribe((data) => {
      console.log(data);
    });
  }
}
```

نلاحظ لدينا Observable واحد يحتوي على قيمة رقمية، بحيث يقوم Observer الخاص به بعمل emit لقيمة عشوائية باستخدام الدالة Math.random()، وقمنا ايضاً بعمل أكثر من Subscription لنفس هذا Observable، والآن لنذهب إلى المتصفح لنرى النتيجة:

نلاحظ انه تم عرض قيم مختلفة لكل Subscription مع ان Observable واحد ونفس Observer ولكن بقيم مختلفة، وهذا غير جيد في حالة أردت ان تعمل مشاركة لنفس القيمة مع جميع Subscribers لهذا Observable، ولكن في حال كان احتياجك عزيزي المتعلم ان يكون هنالك قيم مختلفة لكل Subscription فلا ضير من ان تستخدم الطريقة التقليدية.

اما الآن لنقوم بكتابة نفس المثال السابق ولكن باستخدام Subjects، كالتالي:

App.component.ts

```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {

  private subject = new Subject<number>();
```

```
  constructor() {
    // subscription 1
    this.subject.subscribe((data) => {
      console.log(data);
    });

    // subscription
    this.subject.subscribe((data) => {
      console.log(data);
    });

    this.subject.next(Math.random());
  }
}
```

اما النتيجة في المتصفح، فتكون كالتالي:

نلاحظ ان نفس القيمة تم عمل مشاركة لها لجميع Subscribers، ويجدر الاشارة ومن وجهة نظري ان هذه لا تعتبر ميزة بقدر ما تعتبر احتياج، فمتى ما كان احتياجك ان يكون لكل Subscription قيمة مختلفة فاستخدم الطريقة التقليدية، اما في حال كان احتياجك ان تعمل مشاركة لنفس القيمة مع جميع Subscribers في تطبيقك فاستخدم Subjects، كما نستطيع ان نعمل مشاركة للبيانات عن طريق Observable عن طريق بعض الدوال Operators مثل share، ولكن سنؤجل شرحها إلى وصولنا إلى الباب الخاص بهذه الدوال بإذن الله.

بعدما استعرضنا ميزات Subjects لنستعرض الآن بعض استخداماتها:

- على سبيل المثال نستطيع استخدام Subjects لتعامل مع حالة المستخدم من ناحية تسجيل الدخول من عدمه، فمثلاً نستطيع تعريف متغير على انه Subjects من النوع المنطقي boolean وكلما تغيرت حالة المستخدم نقوم بعمل emit سواء true/false وأي جزء في المشروع يتعلق بحالة المستخدم يعمل Subscription لهذا المتغير وفي حال تغيرت قيمة هذا المتغير، نقوم بإجراء بعض الأوامر.
- نفس الطريقة السابقة ولكن يتعلق بصلاحيات المستخدم، فنقوم بإظهار واخفاء بعض أجزاء التطبيق بناءً على حالة متغير من النوع Subjects.
- وايضاً هناك مثال آخر كأن نتحكم بإظهار وإخفاء spinner عند تحميل أي جزء من التطبيق، فبدلاً من ان نضع في كل component، نستطيع انشاء component منفصل وكلما أردنا اظهار او إخفائه نقوم بعمل emit عن طريق متغير من النوع subjects ونعمل subscription واحد في component الموجود فيه spinner.

وفي الحقيقة الأمثلة والاستخدامات كثيرة، ولكن هذه بعض الحالات التي وردت إلى ذهني اثناء كتابتي لهذا الكتاب، وبعدما تكلمنا بإسهاب عن هذه الميزة المهمة، سوف نتكلم عن أنواع Subjects بشيء من التفصيل.

ملاحظة مهمة: اعلم عزيزي المتعلم ان جميع أنواع Subjects تدور في فلك هل يوجد قيم تم عملها emit قبل أي Subscription ام لا؟ وإذا كان هنالك قيمة هل سيتم قراءتها لهذا Subscription أم لا؟ بحيث هنالك أنواع لا تقوم بقراءة أي قيمة قبل أي Subscription و إنما تقرأ أي قيمة تم عملها emit بعده، وهنالك أنواع أخرى تقوم بقراءة جميع القيم قبل أي subscription وهنالك دوال أخرى تقوم بقراءة آخر قيمة تم عملها emit قبل أي Subscription، اما القيم التي تأتي بعد أي Subscription فجميع الأنواع متشابهة حيث تقوم جميعها بقراءة هذه القيم لأي قيمة بعد أي Subscription.

8-1- Subject:

نستطيع ان نقول ان Subject هو Pure من Subjects، بمعنى انه يمتلك المميزات التي ذكرتها سابقاً والتي تعتبر مشتركة بين جميع أنواع Subjects الأخرى. مثل انه يمتلك القدرة على ان يعمل emit من خارج Observable، ...الخ.

كما ان أي Subscription يتم على هذا Subject فإنه سوف يقرأ القيم من بعد هذا Subscription ولا يحفظ أي قيمة تم عمل لها emit قبله، فمثلاً لو افترضنا انك عزيزي المتعلم قمت بإنشاء متغير على انه Subject، ومن ثم قمت بعمل له Subscription في AppComponent لكي تستطيع ان تقرأ القيم منه، وفي جزء معين من تطبيقك احتجت إلى البيانات التي يعمل لها emit هذا المتغير لذلك قمت ايضاً بعمل Subscription ثاني لكي تتلقى البيانات والقيم من هذا المتغير، وجعلته يبدأ مثلاً عندما يقوم مستخدم تطبيقك بالضغط على زر معين لينتقل إلى هذا الجزء من تطبيقك، ففي هذه الحالة فإن هذا Subscription الثاني لن يقرأ أي بيانات تم عمل لها emit من قبل، وانما سيقرأ القيم بعد هذا Subscription.

لذلك عزيزي المتعلم في حالة أنك في تطبيقك لا تهتم للقيم السابقة لأي Subscription جديد يتم على Observable فإن هذا Subject هو الحل الأفضل لك.

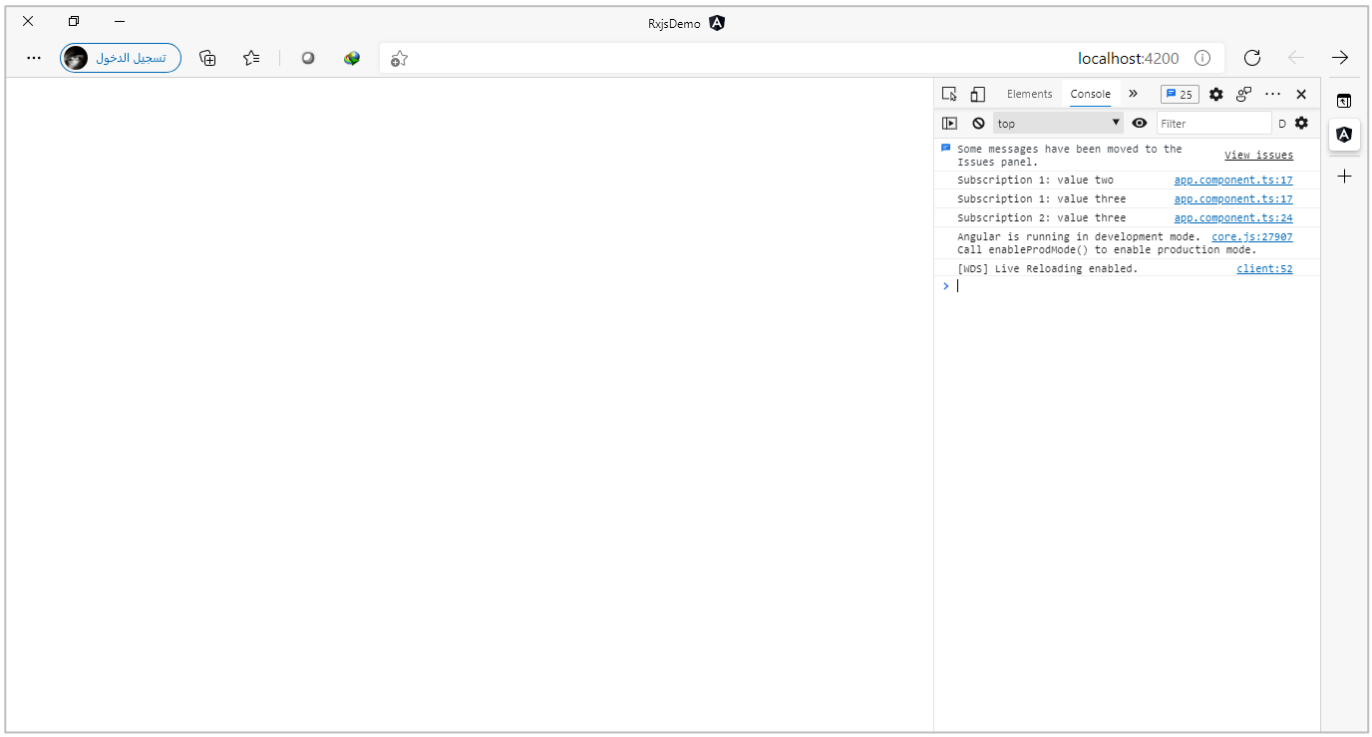
ولنعطي مثال لتوضيح، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new Subject<string>();

  constructor() {
```

```
    // emit first value
    this.subject.next('value one');

    // subscription 1
    this.subject.subscribe((value) => {
      console.log('Subscription 1:', value);
    });

    // emit second value
    this.subject.next('value two');

    // subscription 2
    this.subject.subscribe((value) => {
      console.log('Subscription 2:', value);
    });

    // emit third value
    this.subject.next('value three');
  }
}
```

نلاحظ لدينا اثنين Subscription وقمنا بعمل emit لمجموعة من البيانات، مرة قبل Subscription الأول، والمرة الثانية بعد Subscription الأول والمرة الثالثة بعد Subscription الأول والثاني، ولنرى النتيجة في المتصفح:



نلاحظ عزيزي المتعلم ان اول subscription لم يقرأ القيمة الأولى لأنه تم عمل emit لهذه القيمة قبل ان يتم تفعيل هذا subscription، بينما قرأ القيمة الثانية والثالثة لأنه تم عمل emit لها بعد تفعيل وتشغيل هذا Subscription، ونفس الوضع مع Subscription الثاني حيث لم يتلقى سوى القيم التي تم عمل emit لها بعده.

وبذلك نستطيع ان نقول اننا قمنا بتغطية اهم المفاهيم المتعلقة بهذا النوع من Subjects، وسوف ننتقل الآن إلى النوع الثاني وهو ReplaySubject.

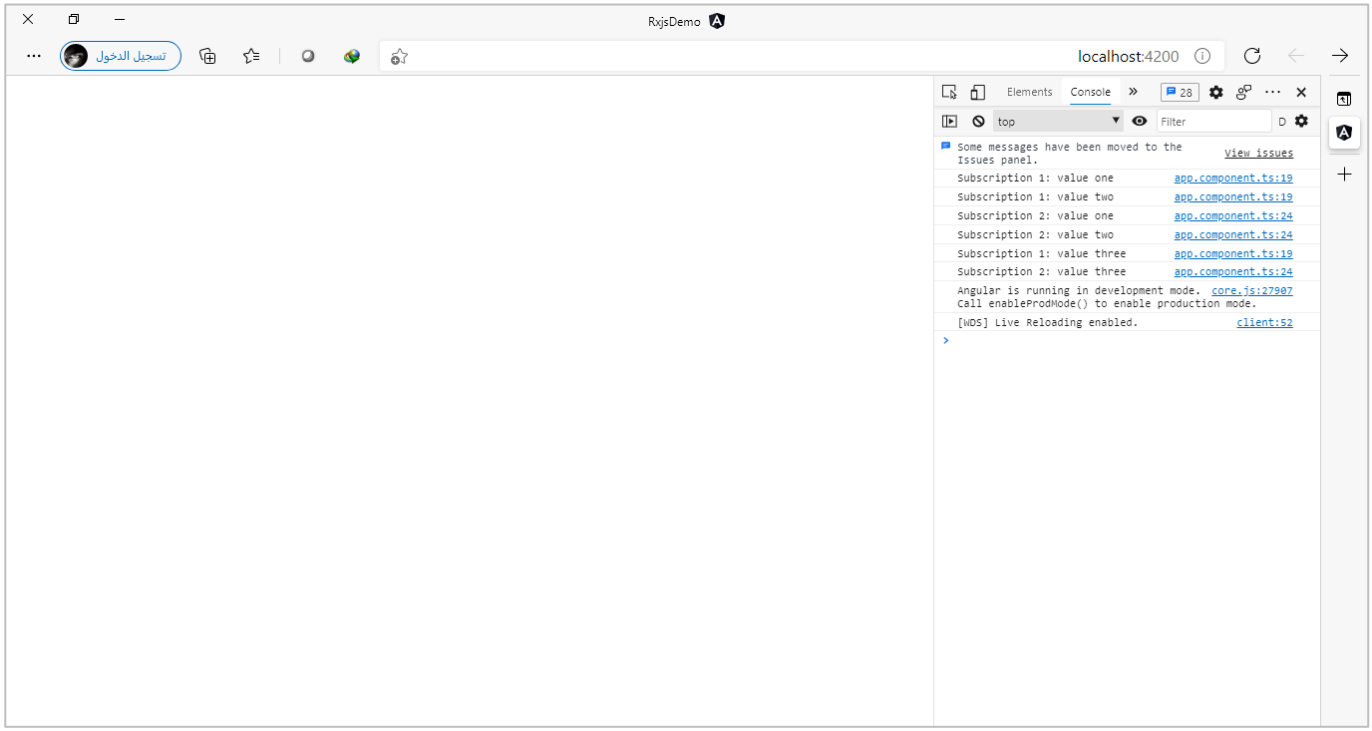
ReplaySubject -2-8:

وهو بعكس النوع السابق يقوم هذا النوع بإرسال جميع القيم السابقة التي تم عمل لها emit قبل أي Subscription جديد ومن ثم يقوم بعمل emit بشكل عادي لكل subscription، ولتوضيح لنقوم بتطبيق نفس المثال السابق ولكن باستخدام ReplaySubject وإجراء بعض التعديلات البسيطة في Logic، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { ReplaySubject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new ReplaySubject<string>();

  constructor() {
    this.subject.next('value one');
    this.subject.next('value two');

    // subscription 1
    this.subject.subscribe((value) => {
      console.log('Subscription 1:', value);
    });

    // subscription 2
    this.subject.subscribe((value) => {
      console.log('Subscription 2:', value);
    });

    this.subject.next('value three');
  }
}
```

وكما نلاحظ العديل الذي قمنا به هو اننا عملنا emit لقيمتين قبل جميع Subscriptions ومن ثم بعد كِلا الـSubscriptions قمنا بعمل emit لقيمة ثالثة، والآن لنرى النتيجة في المتصفح:



نلاحظ قام في Subscription الأول بقراءة جميع القيم التي تم عمل emit لها قبل هذا Subscription، ومن ثم في Subscription الثاني قام بتطبيق نفس الخطوات، وبعدما انتهى من جميع القيم السابقة، وجد ان هنالك قيمة أخرى عمل لها emit بعد هذين Subscription لذلك قام بقراءة هذه القيمة في Subscription الأول ومن ثم قام بقراءتها في Subscription الثاني.

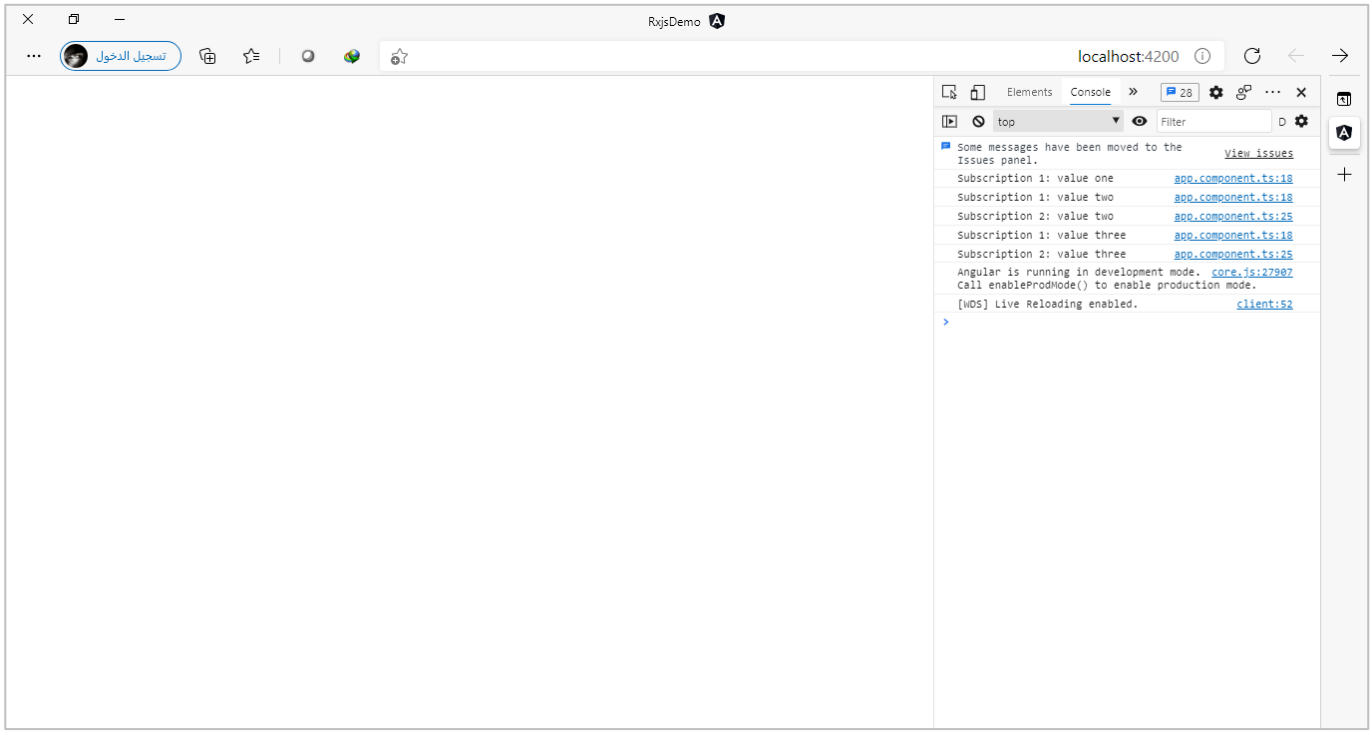
كما نستطيع ان نُحدد للReplaySubject عدد القيم التي سبقت Subscription ونُريد ان نمررها له، فمثلاً قمنا بعمل emit للقيمة "value one" ومن ثم قمنا بعمل subscribe ومن ثم قمنا بعمل emit للقيمة "value two" ومن ثم عملنا subscribe ثاني وأخيرا عملنا emit للقيمة "value three"، فإنه في الوضع الطبيعي في ReplaySubject سوف يقوم بعمل قراءة لجميع القيم التي تم عملها emit لجميع Subscriptions، كما في المثال السابق، ولكن ماذا لو نريد في حالة كان هنالك قيم نُريد فقط آخر قيمة سبقت هذا Subscription هي التي نريد قراءتها فقط، اما الباقي فيتم اهمالها، ففي هذه الحالة يصبح الكود السابق، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { ReplaySubject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private subject = new ReplaySubject<string>(1);

  constructor() {
```

```
    this.subject.next('value zero');
    this.subject.next('value one');

    // subscription 1
    this.subject.subscribe((value) => {
      console.log('Subscription 1:', value);
    });

    this.subject.next('value two');

    // subscription 2
    this.subject.subscribe((value) => {
      console.log('Subscription 2:', value);
    });

    this.subject.next('value three');
  }
}
```

اما النتيجة في المتصفح، فتكون كالتالي:



نلاحظ انه تم قراءة قيمة واحدة فقط للقيم التي سبقت أي Subscription، بحيث ان Subscription الأول سبقته قيمتين هما value zero وvalue one ولكن تم قراءة قيمة واحدة فقط وهي القيمة الأخيرة value one، ومن ثم أتت القيمة value two بعد هذا Subscription وقام بقراءتها ايضاً، ومن ثم تم تفعيل Subscription الثاني وكما نرى سبقه ثلاث قيم هم value zero, value one and value two، وفي الوضع العادي فإن ReplaySubject سوف يقوم بقراءة جميع القيم السابقة ولكن بما اننا مررنا رقم 1 له قرأ فقط قيمة واحدة وهي آخر قيمة value two، وبعدها قام بقراءة القيمة الأخيرة لكلا Subscriptions الأول والثاني بشكل اعتيادي لأنها أتت بعدهما.

وهذا الرقم الذي قمنا بتمريره ليس ثابت فنستطيع تمرير أي رقم نريده بحيث يُعبر عن عدد القيم التي نُريد ان يقرئها ReplaySubject.

لذلك عزيزي المتعلم إذا كنت مهتم بتطبيقك بإن أي Subscription لابد أن يتلقى جميع القيم او بعضها من هذا Observable فعندئذً هذا النوع قد يكون الحل الأفضل لك.

3-8- AsyncSubject:

اما هذا النوع فهو يقوم بقراءة آخر قيمة تم عمل لها emit لأي Subscription سواء تم عمل emit لهذه القيمة قبل او بعد هذا subscription، ولكن يُشترط ان يتم تأشير هذا Observable على انه complete لكي يتم قراءة هذه القيم.

ولنقوم بتطبيق نفس المثال السابق ولكن باستخدام هذا النوع، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { AsyncSubject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new AsyncSubject<string>();

  constructor() {
    this.subject.next('value one');

    // subscription 1
    this.subject.subscribe((value) => {
      console.log('Subscription 1:', value);
    });

    // subscription 2
    this.subject.subscribe((value) => {
      console.log('Subscription 2:', value);
    });

    this.subject.complete();
  }
}
```

نلاحظ لابد ان نُؤشر هذا Observable على انه complete، لكي يتم قراءة آخر قيمة تم عملها emit، سواء كانت قبل او بعد Subscription، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:

نلاحظ قام بقراءة آخر قيمة تم عمل لها emit في كِلا Subscriptions.

لذلك عزيزي المتعلم إذا كان احتياجك ان لا يتم قراءة أي قيمة في Observable إلا إذا تم الانتهاء من هذا Observable وايضاً مهتم بآخر قيمة فقط، فلعل هذا النوع هو الذي تُريده.

8-4- BehaviorSubject:

وهذا النوع قد يُعتبر من أشهر الأنواع وأكثرها استخداماً، وهو مشابهة تماماً لنوع ReplaySubject(1) في حال قمنا بتمرير لها رقم 1، بحيث تقوم بقراءة آخر قيمة تم عمل لها emit في حال وجودها قبل أي Subscription، والفرق الوحيد انها هنا تتميز عن باقي الأنواع بانها تقبل قيمة مبدئية بشكل الزامي على شكل بارامتر.

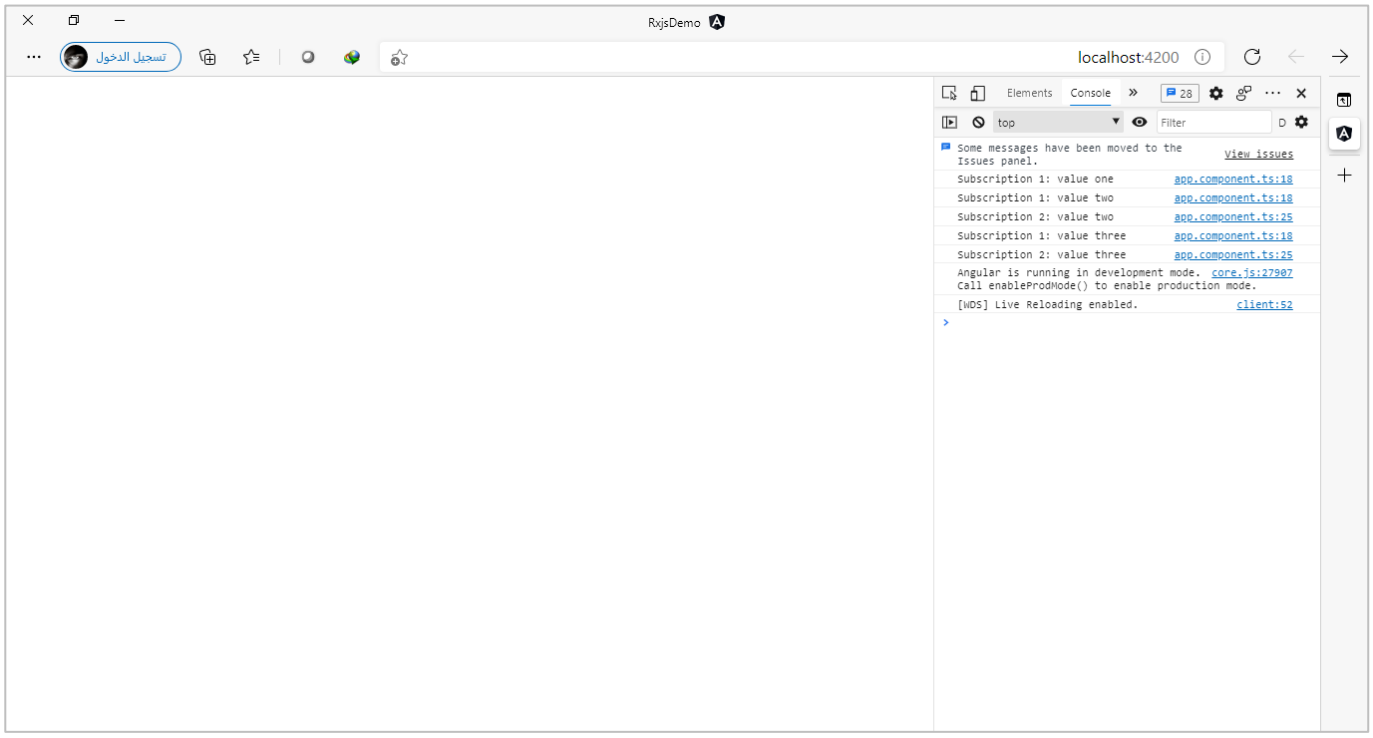
ولتوضيح لنقوم بتطبيق نفس المثال السابق ولكن باستخدام هذا النوع، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new BehaviorSubject<string>('value one');

  constructor() {

    // subscription 1
    this.subject.subscribe((value) => {
      console.log('Subscription 1:', value);
    });

    this.subject.next('value two');

    // subscription 2
    this.subject.subscribe((value) => {
      console.log('Subscription 2:', value);
    });

    this.subject.next('value three');
```

```
  }
}
```

اما النتيجة في المتصفح، فتكون كالتالي:



نلاحظ ان النتيجة مشابهة للمثال في النوع السابق ReplaySubject عندما قمنا بتمرير له الرقم 1، اما لشرح الخطوات التي حدثت في هذا النوع، فهي في البداية عرفنا متغير من النوع BehaviorSubject ومررنا لها قيمة نصية value one وبعدها تم تفعيل Subscription وبما انه يوجد قيمة واحدة فقط تم عمل لها emit اثناء عند تعريف هذا المتغير، فلذلك تم قراءتها في هذا Subscription، ومن ثم تم عمل emit لقيمة أخرى وهي value two وايضاً تم قراءتها في Subscription الأول لأن الثاني إلى الآن لم يتم تفعيله، ومن ثم تم تفعيل Subscription الثاني، وعندها يتم البحث هل يوجد قيم تم عمل لها emit قبله ام لا؟ في حال لم يكن هنالك قيم سيتم قراءة القيمة التي تم إعطائها للمتغير في بداية تعريفنا له وهي value one، اما في حال كان هنالك قيم فسيتم أخذ آخر قيمة قبل هذا Subscription الثاني لكي يتم قراءتها، وهو ما حدث فعلاً حيث يوجد قيمة هي value two لذلك تم قراءتها.

وبذلك عزيزي المتعلم تم شرح وتفصيل واستعراض اهم واغلب التقنيات في هذه الأنواع، وفي الجزء التالي نستعرض سوياً بعض الملاحظات التي يجب الانتباه لها عند التعامل مع Subjects.

8-5-ملاحظات يجب الانتباه لها عند التعامل مع Subjects:

هذه الملاحظات ليس بالضرورة ان تكون تحتاج لها او هي الحل الذي قمته يعتبر الحل الأفضل لها او قد تواجهك في مسيرة تعلمك او تعاملك مع مكتبة Rxjs، ولكن أحببت ان اذكرها هنا لأنك عزيزي المتعلم قد تشاهدها في دورة ما او في مقالة في

الانترنت، وقد تسبب لك بعض اللبس البسيط، ولذلك سوف اوضحها لكي يكون هذا الكتاب بإذن الله مرجع شامل وكافي لأي متعلم يريد ان يتمكن من استخدام مكتبة Rxjs بشكل عام ومع Angular على وجه الخصوص.

8-5-1-استخدام الدالة pipe مع Subjects:

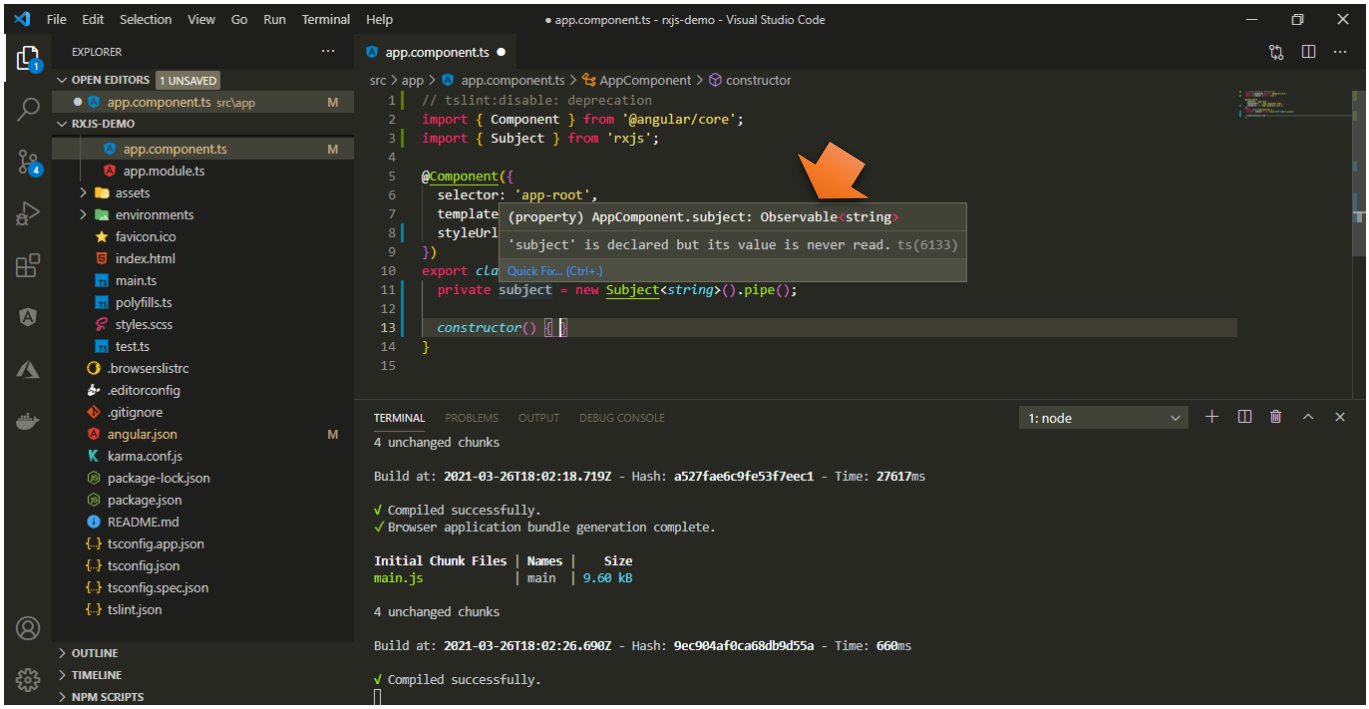
بطبيعة الحال الدالة pipe إلى الآن لم نتطرق لها، ولنا معها وقفات كثيرة ولست مُطالب عزيزي المتعلم ان تعرف ما هي هذه الدالة او استخدامها، وانما ما يهمنا الآن هو عند اضافتها إلى Subjects ما الذي سوف يحدث؟ وهو ما سوف نتحدث عنه في الاسطر القادمة.

عند إضافة الدالة pipe مع Subjects بأنواعها الأربعة في الناتج سوف يصبح Observable عادي، وبذلك تنتفي جميع الميزات التي كان يمتلكها هذا Subjects، كما في المثال التالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new Subject<string>().pipe();

  constructor() { }
}
```

ولو مررنا المؤشر على المتغير subject، سوف يظهر لنا التلميح التالي:



نلاحظ ان النوع الخاص بهذا المتغير أصبح Observable، وهذا غير متوقع لأننا عرفنا هذا المتغير وقمنا بتهيئته على انه أحد أنواع Subjects، إذن ما هو الحل؟ حقيقة الحلول متعددة منها ان نُعرف متغيرين واحد من النوع Subject والآخر من النوع Observable، بحيث إذا أردنا ان نستفيد من مميزات Subject نستخدم المتغير الأول، وإذا أردنا الاستفادة من ناتج هذا Subject مع الدالة pipe فنستخدم المتغير الثاني، كالتالي:

ملف app.component.ts

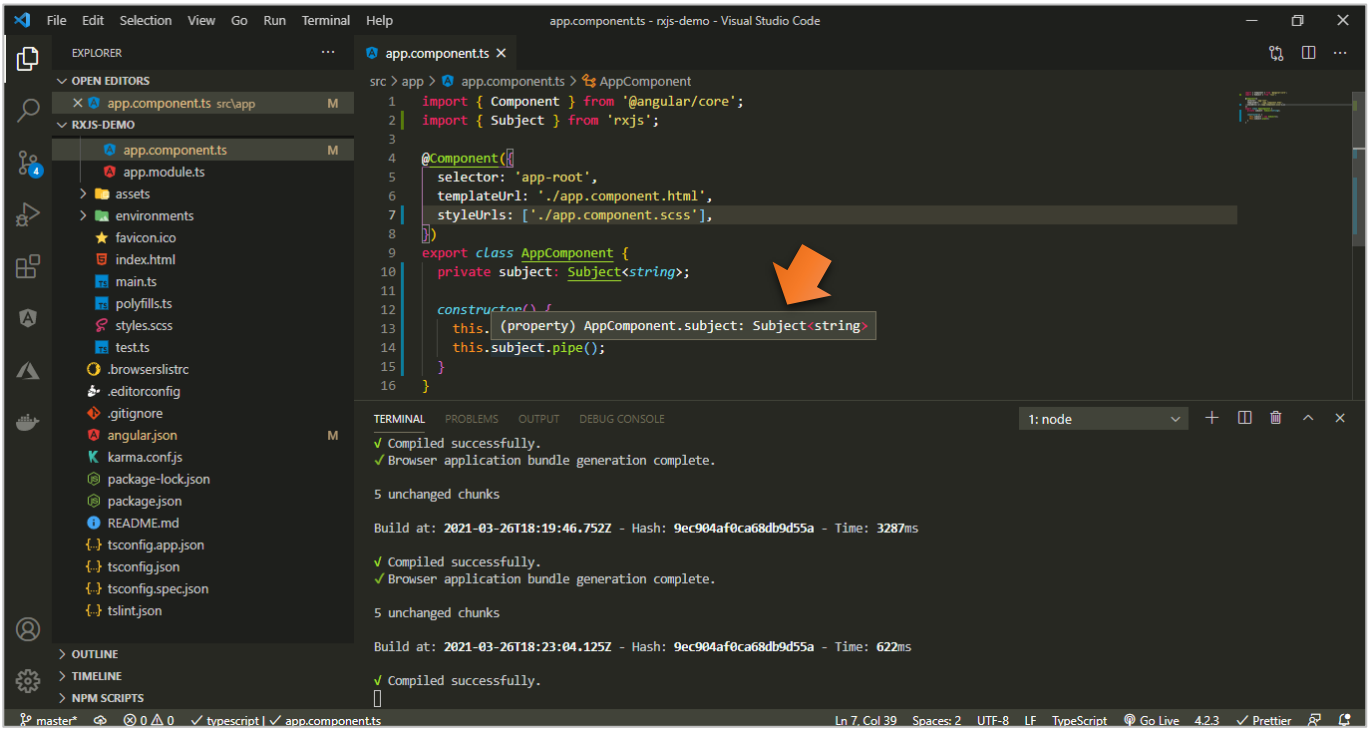
```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new Subject<string>();
  private observable = new Observable<string>();

  constructor() {
    this.observable = this.subject.pipe();
  }
}
```

وهنالك حل آخر وهو المفضل لدي، حيث نقوم بتعريف متغير في سطر برمجي والتهيئة في سطر آخر ومن ثم نستخدم الدالة pipe مع المحافظة على النوع بدون تغييره على انه Observable، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject: Subject<string>;

  constructor() {
    this.subject = new Subject();
    this.subject.pipe();
  }
}
```

ولو وضعنا مؤشر الفأرة فوق المتغير، سوف نجد ان نوعه لم يتغير، كالتالي:



وفي حال أردت لأي سبب من الأسباب ان تقوم بتحويل Subjects إلى Observable مع الاحتفاظ بجميع المميزات الخاصة بالSubjects، فإن القسم التالي قد يكون مناسب لك.

8-2-5-استخدام بعض Operators عوضاً عن Subjects:

اولاً لتحويل أي نوع من أنواع Subjects إلى Observable عادي بدون استخدام الدالة pipe، فنستخدم دالة أخرى اسمها as Observable، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject: Subject<string>;
  private observable: Observable<string>;

  constructor() {
    this.observable = this.subject.asObservable();
  }
}
```

او في سطر واحد، كالتالي:

app.component.ts ملف

```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private subject = new Subject<string>();
  private observable = this.subject.asObservable();

  constructor() {}
}
```

وكما أشرت سابقاً هذا الجزء خاص لنتعلم كيف نستفيد من جميع مميزات Subjects من خلال Observable، وفي الحقيقة نستطيع الاستفادة من جميع مميزات Subjects عن طريق Operators، وتجدر الإشارة انني هنا سوف اذكرها سرداً فقط بدون ذكر أي مثال، والسبب في ذلك هو اننا إلى الآن لم نشرح هذه Operators، ما عدا ان نعمل emit من خارج Observable فهذه الميزة قد شرحت كيفية الاستفادة منها مع Observable سابقاً.

اما الآن لنذكر هذه الدوال Operators، حيث قمت بحصرها، كما في الجدول التالي:

Subjects Type	Subject	AsyncSubject	ReplaySubject	BehaviorSubject
Operators	Share - publish	publishLast	shareReplay - publishReplay	publishBehaviour

9- Hot & Cold Observable:

هذه المصطلحات لقد تعاملنا معها من قبل بشكل عملي، ولكن لم ابين المسميات الخاصة بها، لذلك أحببت ان اشرح هذه المصطلحات لكي قد تجد دورة او مقالة او كتاب أتت على ذكر هذه المصطلحات وتعتقد انها تقنيات جديدة وهي في الأساس من التقنيات والمفاهيم التي ذكرتها سابقا، إذن السؤال المطروح، ما هو Hot Observable؟ Cold Observable؟ والإجابة بسيطة إذا كانت البيانات قد تم عمل لها emit من داخل هذا Observable ففي هذه الحالة تُسمى هذه البيانات بـCold Observable ، اما إذا تم عمل emit من خارج هذا Observable ففي هذه الحالة يتم تسمية هذه البيانات بـHot Observable.

ومما قيل سابقاً نستطيع ان نقول ان Observable العادي يُعتبر Cold اما Subjects بجميع أنواعها تُعتبر Hot. ومن هذا المنطلق نستطيع ان نقول ان Cold Observable لا تعمل emit إلا في حالة طلب Subscription جديد، وهذا بالطبع يحافظ ويضمن ان جميع Subscribers سوف يحصلون على البيانات، بعكس Hot Observable الذي قد يعمل emit

للبيانات عند بداية تعريفنا للمتغير كما هو الحال في BehaviorSubject وبطبيعة الحال قد يتم فقد البيانات لبعض Subscribers الذين عملوا Subscription بعد ان قام Observer بعمل emit لبعض القيم.

وفي الحقيقة ليس هنالك نوع أفضل من الآخر وانما لكل نوع استخداماته، كما ان Observable العادي (cold) له استخداماته وايضاً Subjects الذي يعتبر (hot) له استخداماته.

وغالباً Cold Observable يُعتبر Unicast اما Hot Observable فيعتبر Multicast، لذلك في الجزء التالي شوف نتطرق لهذين المصطلحين، بشكل مجمل.

10-Unicast & Multicast Observable:

وهذه المصطلحات ايضاً تعاملنا معها مسبقاً، من خلال الأمثلة بدون ذكر مصطلحاتها.

حيث ان Observable العادي يُعتبر Unicast بشكل افتراضي، أي ان لكل Subscription جديد يكون له اتصاله الخاص مع الObserver لكي يقوم بعمل emit للبيانات، سواء كانت نفس البيانات بنفس القيم تم عمل emit لها لأكثر من Subscription او كانت البيانات مختلفة، لكن يبقى الفاصل الأساسي ان لكل Subscription مساره او اتصاله الخاص به مع Observer.

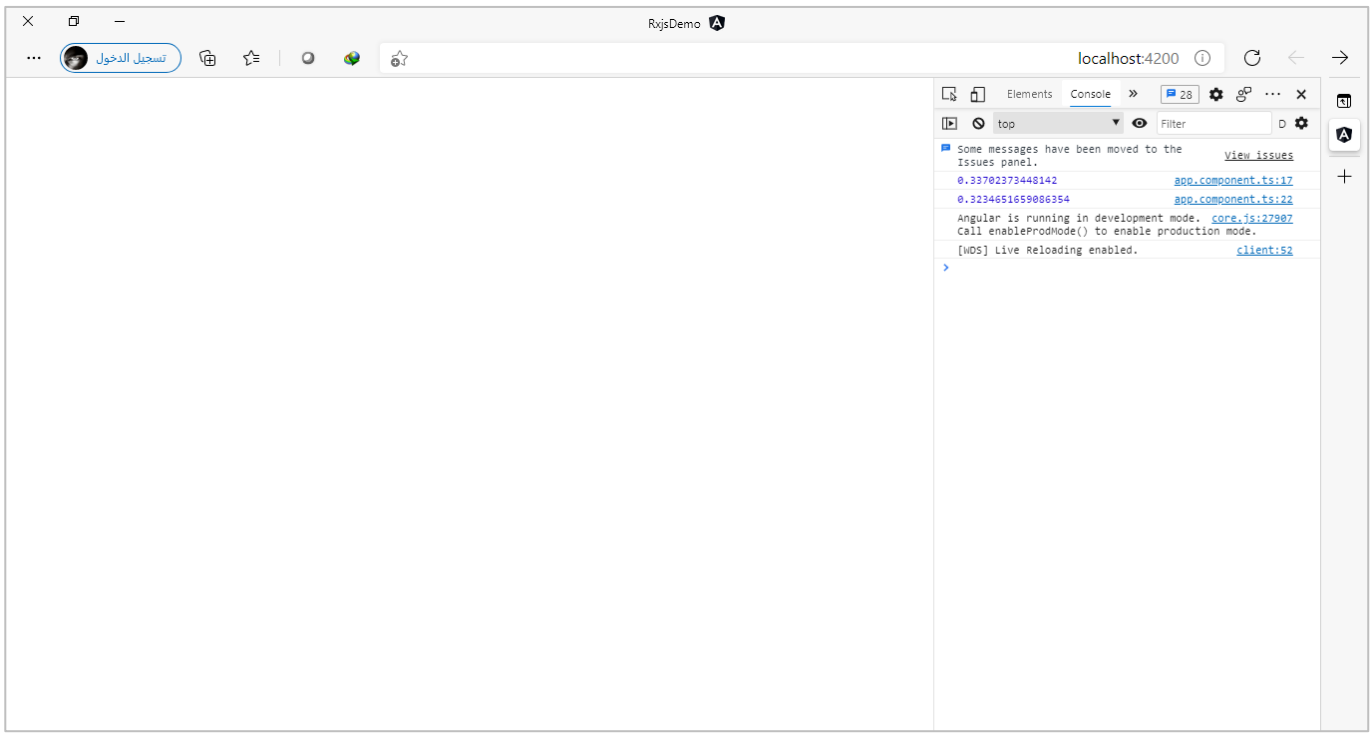
الUnicast تُعتبر Cold Observable أي ان هذه البيانات تم عمل لها emit من داخل هذا Observable عن طريق Observer، ولتوضيح لنُعيد نفس المثال الذي ذكرناه سابقاً في ثنايا هذا الكتاب، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private observable = new Observable((observer) => {
    observer.next(Math.random());
  });

  constructor() {
    // subscription 1
    this.observable.subscribe((data) => {
      console.log(data);
    });

    // subscription 2
    this.observable.subscribe((data) => {
      console.log(data);
```

```
    });
  }
}
```

كما نلاحظ انه تم عمل emit للبيانات من دخال هذا Observable عن طريق Observer الخاص به، وبنفس الوقت قمنا بعمل اثنين Subscription لهذا Observable لقراءة البيانات، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



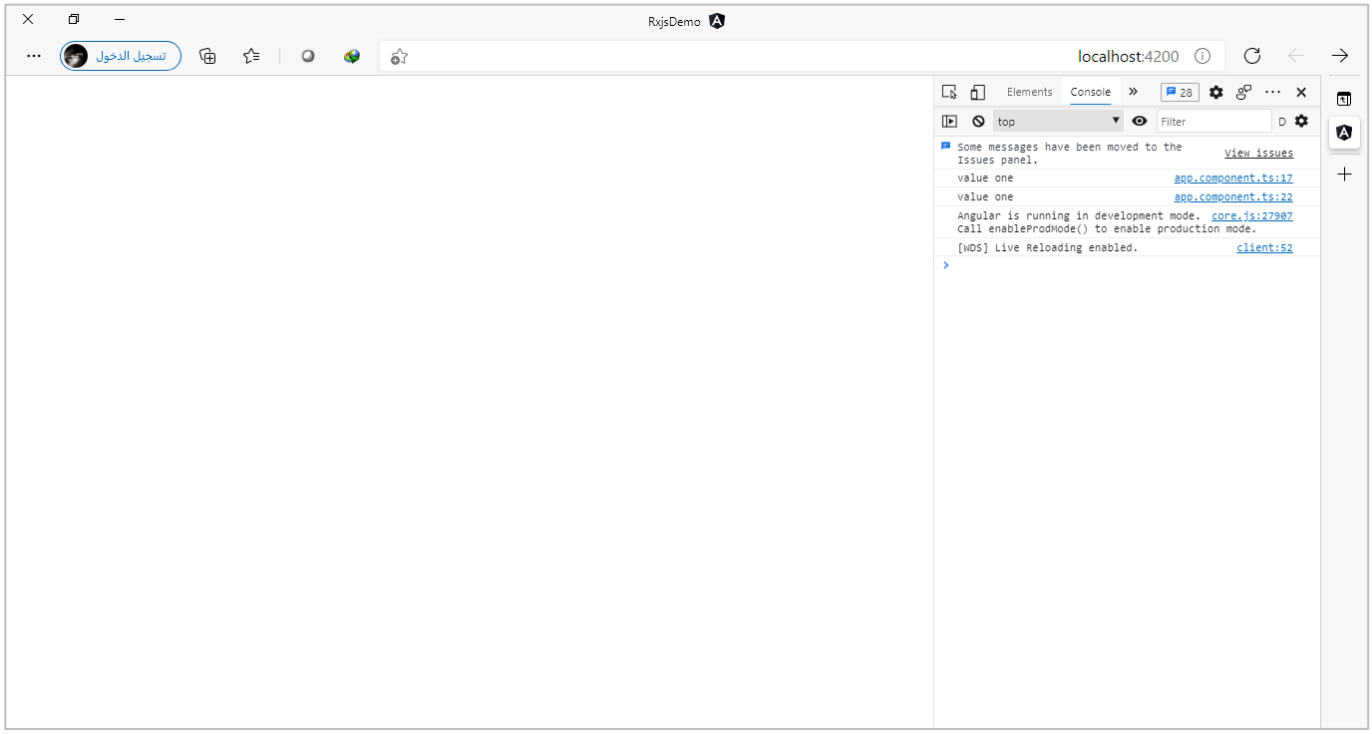
سوف تلاحظ عزيزي المتعلم ان لكل Subscription تم عمل emit لقيمة له، وكما أشرت سابقاً ليس المقصد بالUnicast هو اختلاف القيم لكل Subscription، وانما المقصد هو لكل Subscription اتصاله ومساره الخاص مع هذا Observable سواء تشابهت البيانات ام لم تتشابه، ولتوضيح لنعطي مثال آخر يقوم بعمل emit لنفس البيانات لكل Superscription، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  private observable = new Observable((observer) => {
    observer.next('value one');
  });

  constructor() {
    this.observable.subscribe((data) => {
```

```
      console.log(data);
    });

    this.observable.subscribe((data) => {
      console.log(data);
    });
  }
}
```

والنتيجة في المتصفح، ستصبح كالتالي:



نلاحظ ان نفس القيمة تم عمل emit لها لكلا Subscriptions، ورغم ذلك يعتبر Unicast لأنه في الخلفية قام بعمل اتصال لكل Subscription، وهذا الامر ليس عيباً بالعكس له استخداماته منها ان يضمن ان لكل subscription ان تصله البيانات ولا يفقد البيانات السابقة له قبل أي حدوث هذا Subscription وهو ما يسمى Cold Observable.

وهذا بعكس Multicast والذي يُمثله في الغالب Subjects حيث انه يعمل اتصال واحد لهذا Observable لكل Subscription سواء تشابهت البيانات التي تم عمل emit لها او اختلفت، ولكنها تشترك بنفس المسار وهو ما يسمى ايضاً Hot Observable.

وايضاً Multicast لها استخداماتها، فمن استخداماتها هو عن اتصالك بالسيرفر عن طريق تقنية WebSocket، فلا نُريد ان نستهلك جميع موارد الجهاز لدى مستخدمي تطبيقنا بإن نجعل لكل Subscription اتصاله الخاص وانما نُريد اتصال واحد في السيرفر لجلب البيانات، ومن ثم يتم عمل مشاركة share لهذه البيانات لجميع Subscriptions داخل تطبيقنا من خلال اتصال موحد.

ولكن قد يُعيبه ان عند تفعيل بعض Subscriptions الجديدة قد تفقد البيانات التي تم عمل emit لها قبل حدوث هذا Subscription، كما رأينا عندما تكلمنا عن Subjects.

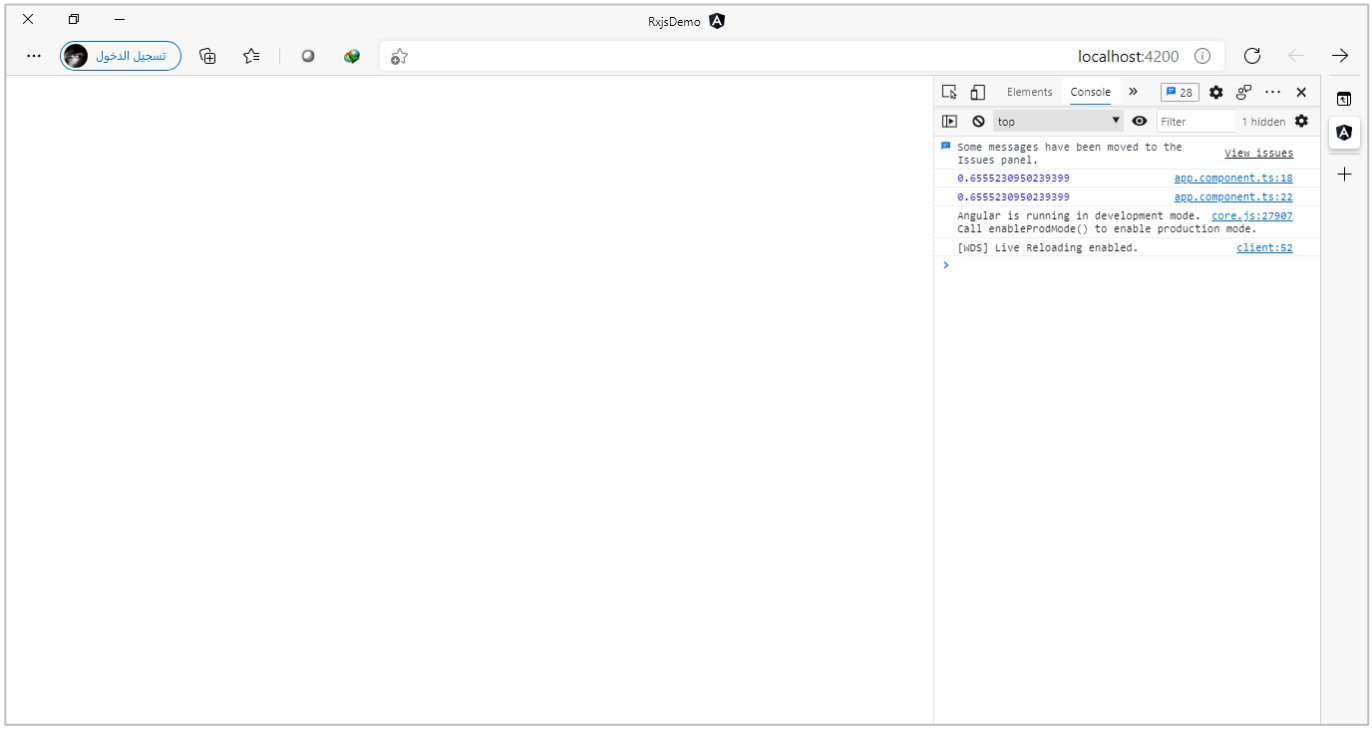
كما نستطيع تحويل Unicast إلى Multicast بعدة طرق منها ان نستخدم Subjects بدلاً من Observable العادي، وفي حال الرغبة باستخدام Observable العادي، نستطيع استخدام بعض الدوال Operators مثل share او publish.

وهنالك بعض المقالات تستخدم طرق أخرى لتحويل Unicast إلى Multicast، ولكنها غير محببة ولا يُفضل استخدامها، مثل هذا المثال:

app.component.ts ملف

```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private observable = new Observable<number>((observer) => {
    observer.next(Math.random());
  });

  private subject = new Subject<number>();

  constructor() {
    this.subject.subscribe((data) => {
      console.log(data);
    });

    this.subject.subscribe((data) => {
      console.log(data);
    });

    this.observable.subscribe(this.subject);
  }
}
```

والنتيجة في المتصفح، كالتالي:



نلاحظ تم تحويل Unicast إلى Multicast بحيث أصبح لدينا اتصال واحد لكل subscription، ولكن كما قلت سابقاً ان هذا الامر غير محبب، ويفضل استخدام طرق أخرى نتكلم عنها في حينها بإذن الله.

11- استخدامات مكتبة Rxjs في عالم Angular:

كل مطور Angular لابد ان يتقن التعامل مع مكتبة Rxjs، فهذه المكتبة وتقنياتها مُستخدمة بكثرة في أطار العمل Angular، يكفي ان تعلم انها مبنية ضمناً بنفس أطار العمل ولا تحتاج إلى أي تحميل وتثبيت، فقط قم بإنشاء مشروع جديد وسوف يتم تضمين هذه المكتبة في مشروعك بشكل تلقائي.

وتجدُر الاشارة ان تعامل Angular في الغالب يختلف قليلاً، عن التعامل مع مكتبة Rxjs مباشرة من حيث عدة نواحي أهمها أن في Angular في الغالب لا نقوم ببناء Observable من الصفر او نعمل emit للبيانات عن طريق Observer، فهذا الجزء يتكفل فيه Angular بالنيابة عنا، والذي يُهمنا كمطوري Angular هو كيفية قراءة هذه البيانات عن طريق Subscription وقبلها كيفية التلاعب وتعديل هذه البيانات بما ويتناسب واحتياجاتنا عن طريق الدالة pipe وما تحتويه من دوال Operators.

فمثلاً عندما نتعامل مع Forms او Router او HttpClient (سوف نُفرد لها فصل كامل من هذا الكتاب بإذن الله) او EventEmitter، Async Pipe، ...الخ.

وقد تعاملنا في هذه السلسلة من خلال الكتب الخمس السابقة، في أكثر من موضع مع Observables سواء عند تعاملنا مع Routing وتقنياته المختلفة او مع Forms او حتى مع Components. وتستطيع الرجوع عزيزي المتعلم إلى هذه الكتب من هذه السلسلة للاستزادة أكثر.

وفي هذا الكتاب ايضاً سوف اتطرق إلى بعض الأمثلة للتعامل مع مكتبة Rxjs مع Angular، ولعلنا نُؤجل الكلام بها بعدما نتطرق إلى Operators في الفصل التالي وبالتحديد في الفصل الخاص بـHttpClient.

12-معالجة الأخطاء بشكله البسيط عن طريق Angular وتقنيات Observable:

معالجة الأخطاء من الأمور الهامة التي يجب على كل مطور اجادتها ومعرفة طرق التعامل معها، فمثلاً عند وجود خطأ في Observable سواء كان من السيرفر او من اتصال الانترنت لدى مستخدم التطبيق او أي خطأ آخر فعندئذً لابد ان نعالج هذا الخطأ سواء عن طريق اظهار رسالة للمستخدم او محاولة معالجة هذا الخطأ بشكل تلقائي وغيره من الأمور الأخرى، وتتيح لنا مكتبة Rxjs عدة طرق للتعامل مع الإخطاء، وفي هذا الجزء سوف نتطرق إلى الشكل البسيط للتعامل مع الإخطاء، اما الطرق المتقدمة فنحتاج إلى معرفة بعض التقنيات والميزات الأخرى قبل التطرق لها، لذلك سنُؤجل الكلام فيها إلى حينها.

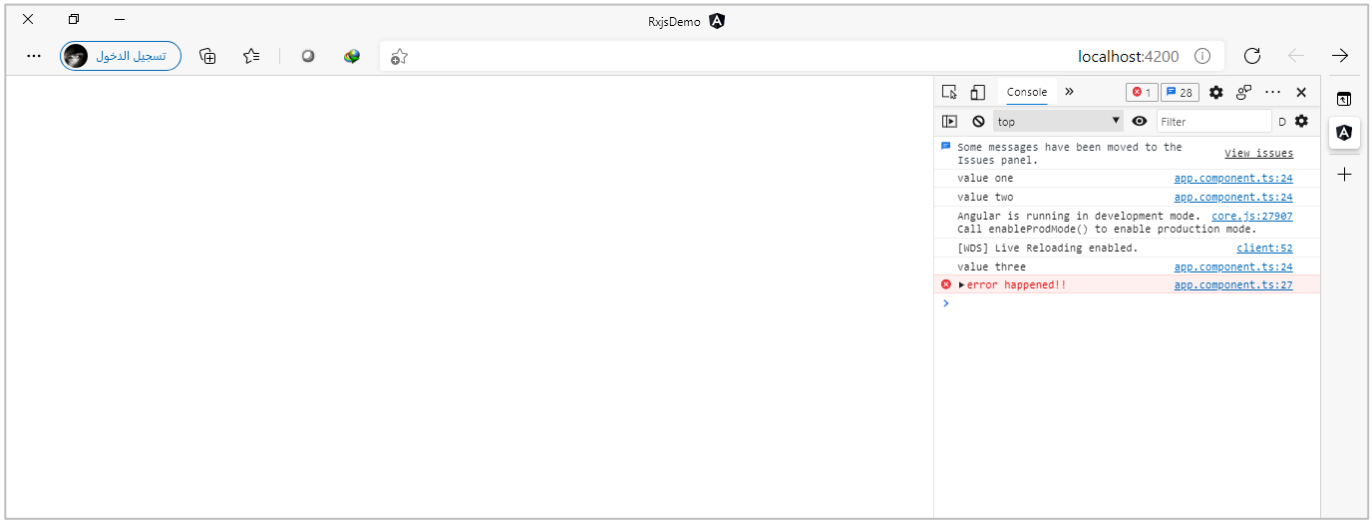
اما الآن لنعطي مثال على التعامل مع الأخطاء بشكله البسيط، وسوف نقوم بمعالجة الخطأ عن طريق Subscriber، حيث كما قلنا سابقاً انه هو الآخر يمتلك ثلاث دوال مشابهة للدوال الخاصة بـObserver وهي next لقراءة البيانات وerror لمعالجة الأخطاء وcomplete في حال اكتمال هذا Observable. ولتوضيح لنشاهد المثال التالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {

  private observable = new Observable((observer) => {
    observer.next('value one');
    observer.next('value two');
    setTimeout(() => {
      observer.next('value three');
    }, 1000);
    setTimeout(() => {
      observer.error('error happened!!');
    }, 2000);
  });

  constructor() {
    this.observable.subscribe({
      next: (values) => {
        console.log(values);
      },
      error: (error) => {
```

```
        console.error(error);
      },
    });
  }
}
```

قمنا بعمل emit للبيانات عن طريق Observer ومن ثم بعد ثانيتين قمنا بعمل emit لخطأ معين، وفي Subscription استقبلنا هذا الخطأ في Subscriber عن طريق الدالة error، وكما نلاحظ قمنا بعمل معالجة للخطأ بنفس component، وهذا غير محبب ويفضل جعل component لعرض البيانات فقط، ولكن كما أشرت سابقاً ان هذه الطريقة البسيطة للتعامل مع الأخطاء، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ تم عمل emit للقيمتين الأولى والثانية وتم استقبالها في الدالة next في Subscriber، ومن ثم بعد ثانية تم عمل emit للقيمة الثالثة وايضاً تم استقبالها وعرضها، وأخيراً بعد ثانيتين تم عمل emit للخطأ وعندئذً تم استقبالها هذه المرة في الدالة error في Subscriber ومن ثم تم عرضها في console.

الفصل الثالث

Operators

1- مقدمة:

امن الأمور المهمة التي يجب اجادتها والتعامل معها هي Operators، حيث تقدم لنا هذه المكتبة دوال كثير تقدر بأكثر من 100 دالة تقريباً، تم تصميمها لكي تُسهل على مستخدمي هذه المكتبة مهامهم وملبية لأغلب احتياجاتهم وليس لتعقيد البرمجة عليهم، وهذا التصور قد نجده عند بداية من يُحاول تعلم هذه الدوال ولكن مع مرور الوقت سوف يجد ان هذه الدوال سهلة الاستخدام وتختصر عدد من الاسطر البرمجية من خلال دالة واحدة، واعلم عزيزي المتعلم انه ليس المطلوب منك تعلم جميع هذه الدوال ولكن هنالك دوال مشهورة وكثير الاستخدام وهي التي سوف نقوم بتغطيتها بإذن الله، وفي حال مستقبلاً لم تجد احتياجك من هذه الدوال المشهورة عندها تستطيع مع بعض التعلم الذاتي البسيط تعلم أي دالة أخرى في هذه المكتبة.

ولكن السؤال المطروح ما هي هذه Operators، وبشكل عام نستطيع ان نقول ان هذه Operators هي عبارة عن دوال Methods تستقبل Observable كمُدخل Input وتُخرج Observable كمُخرج Output او بمسمى آخر كنتيجة، وهذا المُخرج (ممكن) ان يكون مُدخل لدالة أخرى وهكذا بحيث يُصبح لدينا سلسلة من الدوال.

وبشكل عام هُنالك نوعين من الدوال وهي Static Operators وPipeable Operators، وسوف نستعرض هذه الأنواع مع شرح أشهر الدوال لها.

2- Static Operators:

وتسمى ايضاً Creations Operators لأن اغلب هذه الدوال يتم استخدامها لإنشاء Observable سواء كان جديد او من سلسلة نصية او رقمية او من مصفوفة او كائن، باختصار أي نوع من البيانات، كما هُنالك دوال يتم استخدامها لدمج أكثر من Observable مع بعضها البعض في Observable واحد.

وهذه الدوال يتم استدعائها من خلال مكتبة rxjs مباشرة وليس من المجلد الفرعي المسمى operators، كالتالي:

import {..., name of operator, ...etc.} from "rxjs"; ✓

import {..., name of operator, ...etc.} from "rxjs/operators"; ✗

وسوف نستعرض أغلب هذه الدوال بالشرح والتوضيح وإعطاء الأمثلة من خلال النقاط التالية:

2-1- from():

هذه الدالة تقوم بإنشاء Observable من البيانات التي تدعم Iteration بشكل ضمني، مثل المصفوفات Arrays او String او دوال Set() وMap() او Generators Function أو ...الخ، بمعنى أي نوع بيانات يسمح او يحتوي على دوال نستطيع عن طريقها ان نعمل Loop على القيم الموجودة بها، وايضاً تقوم بتحويل Promise إلى Observable، وعند الانتهاء من جميع البيانات تقوم بشكل تلقائي بعمل إغلاق لهذا Observable.

ولفهم أعمق تستطيع عزيزي المتعلم الرجوع إلى دورة استاذنا المبدع والمتألق دائما أسامة الزيرو على اليوتيوب بعنوان ECMAScript 6، وحقيقة هذه الدورة سوف تساعدك ليس فقط لمعرفة Iterator وما يتعلق بها وانما جميع المميزات المضافة حديثا في هذا الإصدار من لغة الجافا سكربت مثل (let, const, Arrow Function, Spread Operator, Rest Parameter, Destruction, Generators Function, ...etc.) وغيرها الكثير والتي سوف تساعدك عزيزي المتعلم لفهم أعمق لإطار عمل Angular او أي إطار عمل آخر يعتمد على لغة الجافا سكريبت.

كما انها تأخذ بارامتر واحد وهو البيانات التي نُريد ان ننشأ Observable منه، وبنفس الوقت تُعيد لنا Observable نوعه من نوع البيانات التي قمنا بتحويلها إلى هذا Observable.

بمعنى لو افترضنا اننا نُريد ان نقوم بتحويل بيانات نصية من النوع string إلى Observable، في البداية نتأكد هل هي Iteratable ام لا، وبطبيعة الحال النوع string يدعم iteration بشكل ضمني، أي يحتوي على دوال مبنية ضمناً تسمح بعمل تكرار Loops على البيانات التي يحتويها او نستطيع ان نقول يسمح بشكل عام بعمل تكرار على البيانات المعرفة من هذا النوع، ففي هذه الحالة باستخدام الدالة from سوف يتم انشاء Observable جديد وتُعيد Observable من النوع string أي Observable<string>.

وأخيراً يجب الاشارة إلى ان الدالة from تقوم بعمل emit لكل قيمة على حدا، بمعنى لو كان لدينا مثلاً نوع مثلاً نص string، وقمنا بإنشاء Observable من هذا النوع عن طريق الدالة from، فإنه سوف يقوم بعمل emit في كل مرة حرف او بمعنى اصح خانة، ولا تقلق عزيزي المتعلم سوف أقوم بإعطاء أمثلة على جميع هذه الأنواع وكيفية انشاء Observable منها، والتي سوف تُساعدك بإذن الله لفهم هذه الدالة بشكل أفضل.

اما الآن فسوف نستعرض بعض الأمثلة على بعض الأنواع المشهورة وكيفية تحويلها إلى Observable عن طريق استخدام الدالة from، او بمعنى آخر كيفية انشاء Observable من هذا الأنواع.

2-1-1- إنشاء Observable من المصفوفات Arrays عن طريق الدالة from:

من أشهر الأنواع التي تدعم Iteration هي المصفوفات، عوضاً على انها مستخدمة بشكل كثير في البرمجة، لذلك نستطيع تحويل هذه المصفوفات إلى Observable بشكل سهل عن طريق تمرير المصفوفة إلى هذه الدالة على شكل بارامتر.

وسوف نُبقي على نفس المشروع الذي قمنا بإنشائه في الفصل السابق، مع حذف جميع الأمثلة السابقة ان وجدت.

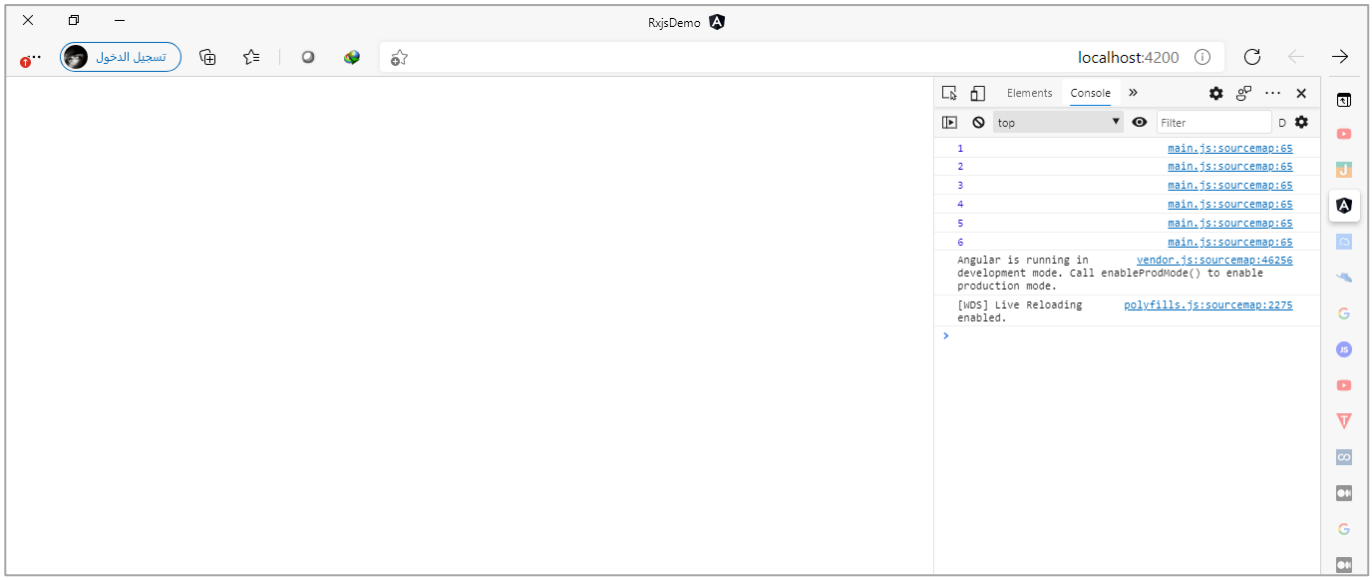
اما الآن لنقوم بإعطاء مثال على كيفية استخدام المصفوفات مع الدالة from، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

```
})

export class AppComponent implements OnInit {
  private obs = from([1, 2, 3, 4, 5, 6]);

  constructor() {
    this.obs.subscribe({
      next: (value) => {
        console.log(value);
      },
    });
  }

  ngOnInit(): void {}
}
```

نلاحظ اننا في البداية قمنا باستدعاء الدالة from من مكتبة rxjs مباشرة وذلك لأنها تعتبر من دوال static كما اشرنا سابقاً، ومن ثم قمنا بتمرير المصفوفة على شكل باراميتر لهذه الدالة لأننا اشرنا سابقاً بإن هذه الدالة تستقبل باراميتر واحد (الزامي)، وبذلك سوف يتم انشاء Observable من هذه المصفوفة حيث قما بتخزينها في متغير اسميته obs، وأخيراً بما ان هذا Observable فلابد ان نعمل Subscribe له لكي نستطيع ان نقرأ البيانات، وهذا الذي عملناه حيث قمنا بطباعة هذه القيم في console، اما الآن لنشاهد النتيجة في المتصفح:



نلاحظ قامت الدالة from بعمل flatting للمصفوفة أي تقسيمها وتجزئتها كل قيمة لوحدها.

كما نستطيع ان نمرر متغير يمثل المصفوفة بدلاً من المصفوفة نفسها، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  private arr = [1, 2, 3, 4, 5, 6];
  private obs = from(this.arr);

  constructor() {
    this.obs.subscribe({
      next: (value) => {
        console.log(value);
      },
    });
  }

  ngOnInit(): void {}
}
```

ولو قمنا بتشغيل التطبيق فسوف تظهر لنا نفس النتيجة في المتصفح.

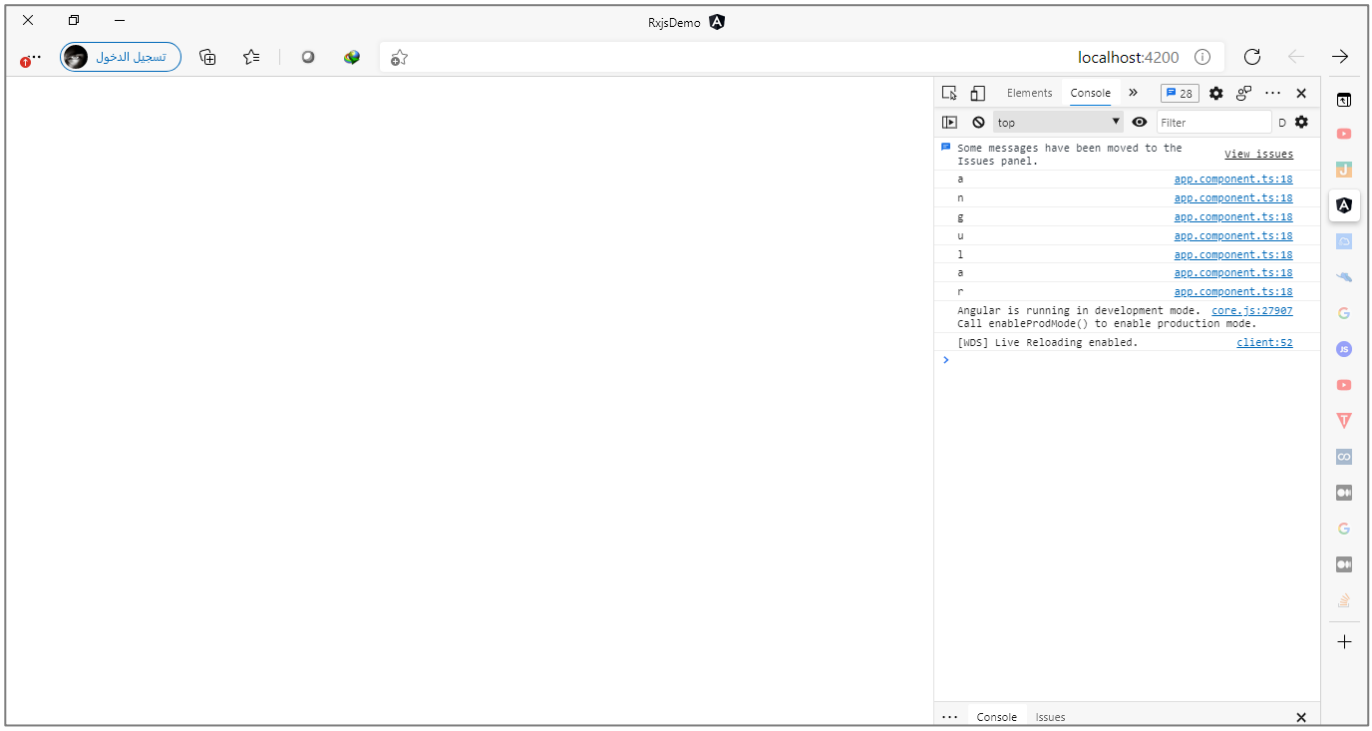
2-1-2- إنشاء Observable من سلسلة نصية string عن طريق الدالة from:

كما فعلنا مع المصفوفات نستطيع هنا ايضاً ان نقوم بتحويل أي نص إلى Observable للاستفادة من الميزات التي يُقدمها، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  private str = 'angular';
  private obs = from(this.str);

  constructor() {
    this.obs.subscribe({
      next: (value) => {
        console.log(value);
      },
    });
  }

  ngOnInit(): void {}
}
```

اما النتيجة في المتصفح، فتكون كالتالي:



نلاحظ تم عمل flatting لهذه السلسلة النصية وتم قراءة كل حرف على حدا، ولعل من خلال هذا المثال تتضح الصورة أكثر من حيث كيف ان الدالة from تقوم بعمل flat للقيم وقراءتها قيمة قيمة.

2-1-3- انشاء Observable من Promises عن طريق الدالة from:

الPromises من التقنيات المهمة في عالم الجافا سكربت، ومستخدمة بكثرة وليس هنا المقام لشرحها، وتستطيع عزيزي المتعلم مراجعة نفس الدورة التي أشرت لها سابقاً للمهندس أسامة الزيرو على اليوتيوب تحت عنوان ECMAScript 6.

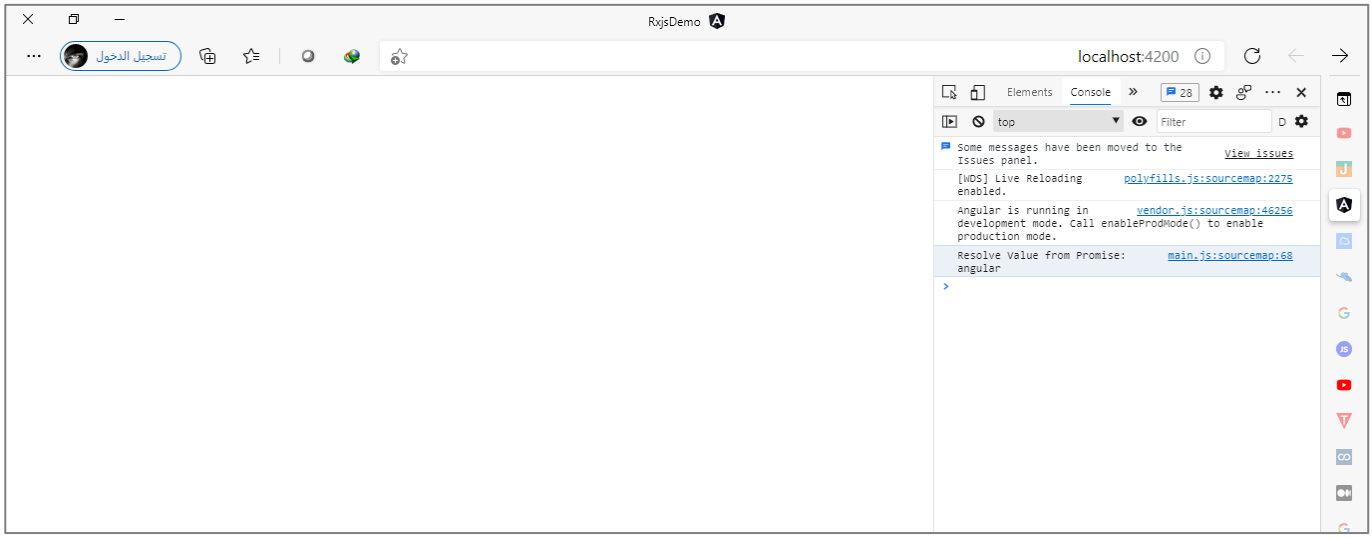
ولو تلاحظ عزيزي المتعلم قد أشرت في الفصل السابق إلى ان تقنيات Observable تقدم مميزات أكثر مما تقدمها Promises عند التعامل مع البيانات الغير تزامنية Async Data، ولذلك في حال كان لديك Promise وأردت ان تقوم بتحويله إلى Observable للاستفادة من جميع الميزات التي تقدمها هذه التقنية، فتستطيع ذلك بكل بساطة عن طريق استخدام الدالة from وتمرر لها Promise الذي تُريده، وسوف تقوم هذه الدالة بالنيابة عنك بتحويل هذا Promise إلى Observable وايضاً قراءة البيانات القادمة من الدالتين resolve أو reject في next بحالة resolve وفي error في حالة reject، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  private promise = new Promise((resolve, reject) => {
    resolve('angular');
  });

  private obs = from(this.promise);

  constructor() {
    this.obs.subscribe({
      next: (value) => {
        console.log('Resolve Value from Promise: ' + value);
      },
      error: (error) => {
        console.error('Reject Value from Promise: ' + error);
      },
    });
  }

  ngOnInit(): void {}
}
```

هذا في حالة resolve سوف يتم قراءة البيانات القادمة منها عن طريق الدالة next، والنتيجة في المتصفح تكون كالتالي:



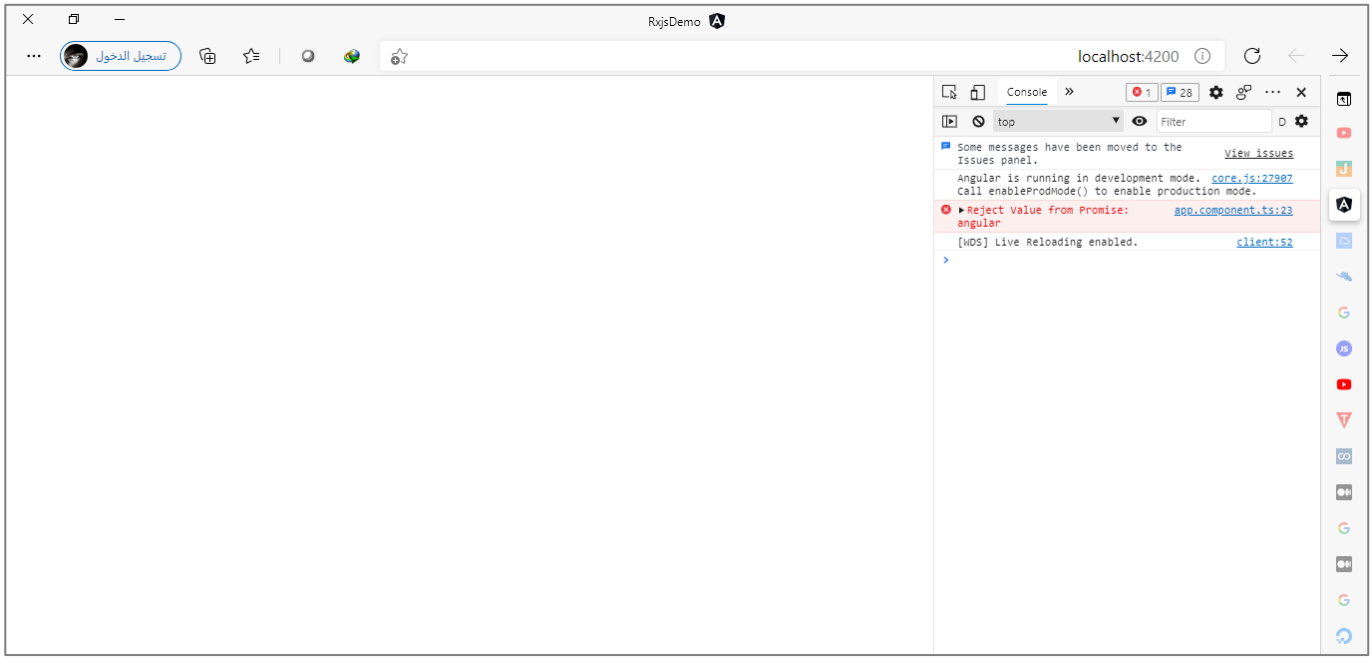
نلاحظ تم قراءة القيمة القادمة من الدالة resolve عن طريق الدالة next في Subscriber، اما في المثال التالي فسوف نشاهد كيف يتم قراءة القيمة من الدالة reject، كالتالي:

app.component.ts ملف

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  private promise = new Promise((resolve, reject) => {
    reject('angular');
  });
  private obs = from(this.promise);

  constructor() {
    this.obs.subscribe({
      next: (value) => {
        console.log('Resolve Value from Promise: ' + value);
      },
      error: (error) => {
        console.error('Reject Value from Promise: ' + error);
      },
    });
  }
  ngOnInit(): void {}
}
```

اما النتيجة في المتصفح، فتكون:



4-1-2- إنشاء Observable من DOM Nodes عن طريق الدالة from:

DOM Nodes بشكلها البسيط هيا عبارة عن عناصر وتاغات HTML بعدما يقوم المتصفح بقراءتها وتشغيلها، حيث يقوم المتصفح عندما يتم تشغيل التطبيق او الموقع فإن المتصفح سوف يقوم بتحويل هذه العناصر بشكل شجري متفرع في ذاكرة الجهاز RAM بحيث كل عنصر HTML يُسمى node.

وبما ان هذه nodes تدعم iteration بشكل ضمني أي بمعنى نستطيع ان نعمل لها Loop فلذلك نستطيع انشاء Observable من هذه nodes باستخدام الدالة from.

وسوف استعرض مثالين الأول لطريقة قراءة DOM Nodes باستخدام الجافا سكريبت بدون استخدام أي اطر عمل والثاني سوف أوضح كيف يتم قراءة هذه nodes بأسلوب Angular.

اما الآن لنعطي المثال الأول ولنفرض انه لدينا تاغ ul وبداخله ثلاث تاغات li، كالتالي:

ملف app.component.html

```
<ul>
  <li>Angular</li>
  <li>React</li>
  <li>Vue</li>
</ul>
```

والخطوة التالية هي قراءة هذه nodes باستخدام أسلوب الجافا سكريبت العادي، ومن ثم انشاء Observable منها واخيراً نعمل لها subscribe لكي نقرا قيم هذا Observable، كالتالي:

ملف app.component.ts

```
import { Component, AfterViewInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements AfterViewInit {

  constructor() {}

  ngAfterViewInit(): void {
    const obs = from(document.querySelectorAll('li'));
    obs.subscribe((element) => {
      console.log(element.textContent);
    });
  }
}
```

نلاحظ اولاً اننا قمنا بكتابة الكود في دالة ngAfterViewInit وهي من دوال Lifecycle Hook والتي يتم تنفيذ محتوياتها بعدما يتم قراءة تاغات وعناصر صفحة HTML وتحويلها إلى nodes في ذاكرة الجهاز.

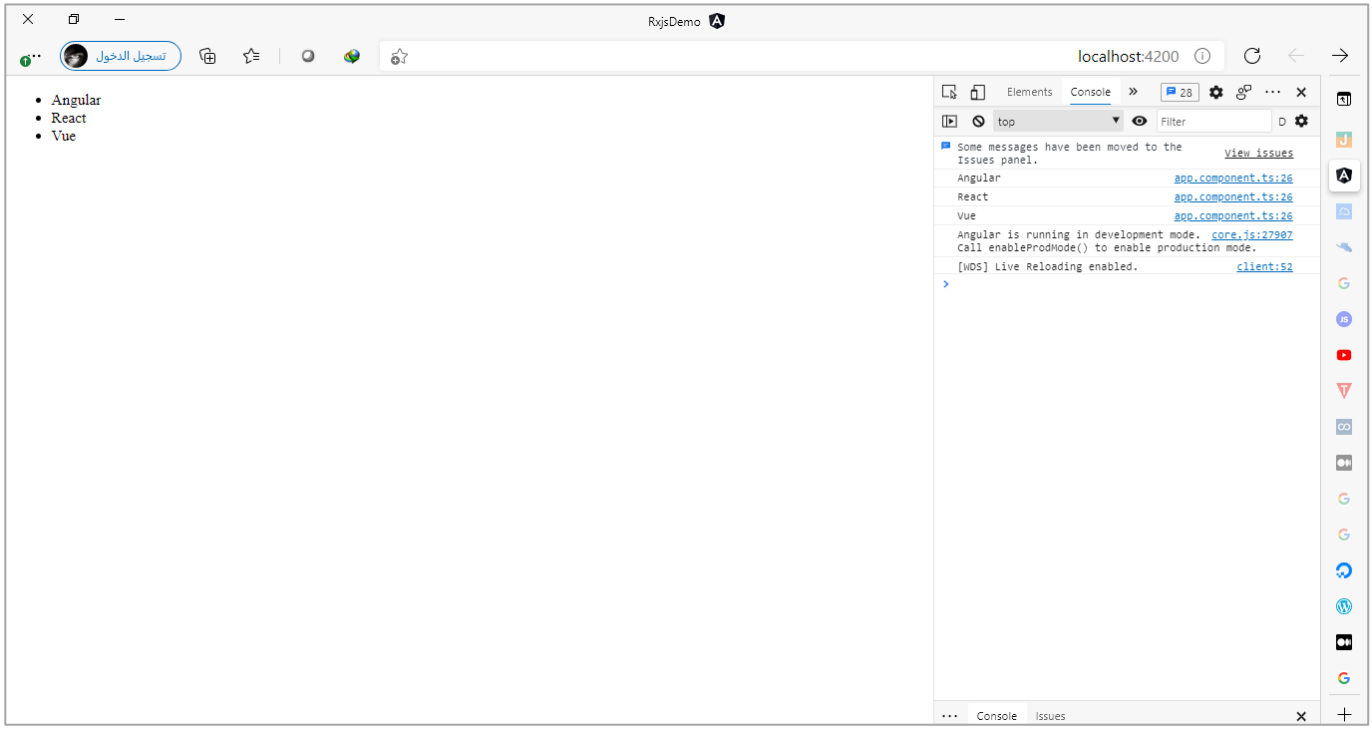
ومن ثم باستخدام دالة querySelectorAll وهي من دوال Vanilla JavaScript والتي تقوم بجلب أي nodes نمرره له وفي حالتنا هذه مررنا له li ومن ثم يقوم بتحويلها مباشرة إلى Observable وتخزين هذا Observable في متغير اسميته obs، وأخيراً لابد ان نعمل Subscribe لكي نستطيع ان نقرأ هذا Streams وهو الذي فعناه في الاسطر اللاحقة.

مع العلم اننا نستطيع ان نعمل subscribe مباشرة، كالتالي:

app.component.ts ملف

```ts
import { Component, AfterViewInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements AfterViewInit {
  constructor() {}
  ngAfterViewInit(): void {
    const obs = from(document.querySelectorAll('li')).subscribe((element) => {
      console.log(element.textContent);
    });
  }
}
```

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



والآن لنقوم بعمل نفس المثال السابق ولكن بطريقة Angular، في حال حدوث أي لبس عليك عزيزي المتعلم تستطيع الرجوع إلى الكتاب الثاني من هذه السلسلة باسم Angular Components and Services حيث قمت بشرح جميع هذه التقنيات بشكل مفصل، ولنرجع إلى المثال ولنرى التغيير، كالتالي:

app.component.html ملف

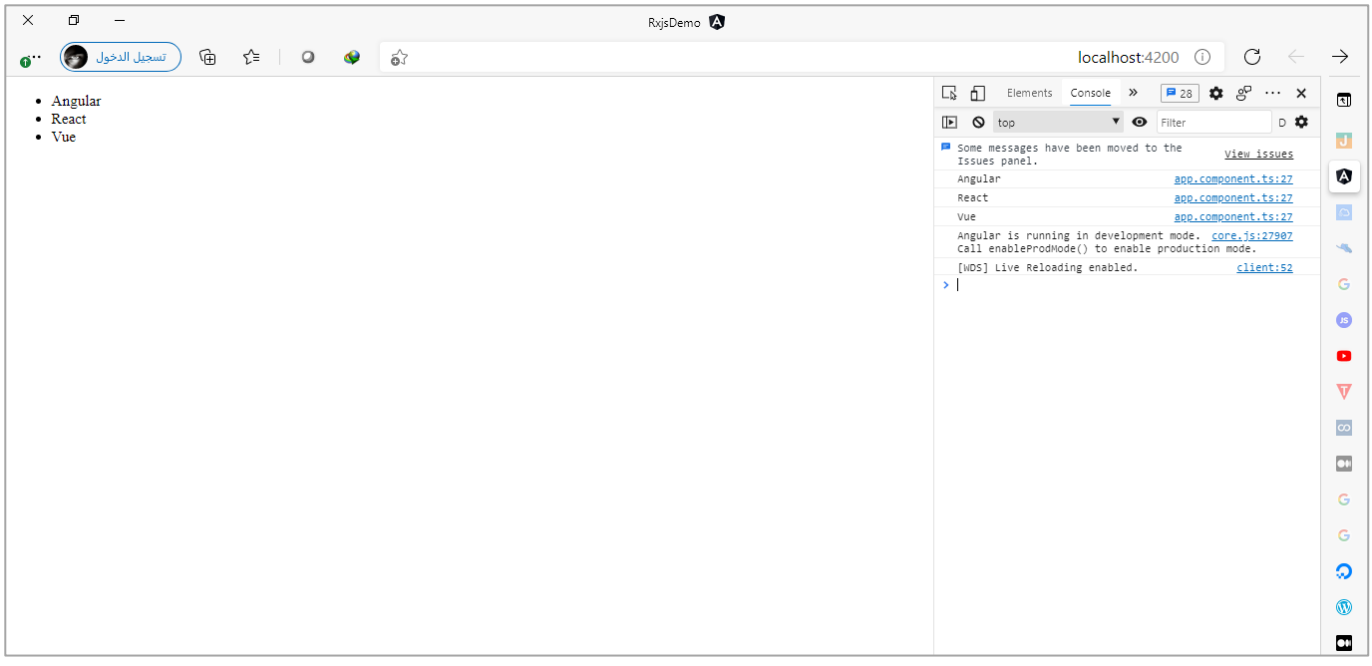
```html
<ul>
  <li #element>Angular</li>
  <li #element>React</li>
  <li #element>Vue</li>
</ul>
```

وفي ملف class نقوم بكتابة الكود التالي:

ملف app.component.ts

```
import {
  Component,
  QueryList,
  ViewChildren,
  AfterViewInit,
  ElementRef,
} from '@angular/core';
import { from, Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements AfterViewInit {
  @ViewChildren('element', { read: ElementRef })
  private elements: QueryList<ElementRef<HTMLLIElement>>;
  private obs: Observable<ElementRef>;

  constructor() {}

  ngAfterViewInit(): void {
    this.obs = from(this.elements.toArray()).subscribe((element) => {
      console.log(element.nativeElement.textContent);
    });
  }
}
```

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



5-1-2- إنشاء Observable من Generators Function عن طريق الدالة from:

وليس هنا المقام لشرح تقنية Generators Function وانصحك عزيزي المتعلم بنفس المصدر السابق وهو دورة أسامة الزيرو على اليوتيوب تحت اسم ES6 حيث شرح بشكل سهل وسلس جميع المفاهيم المتعلقة بهذه التقنية، اما هنا فما يهمنا هو كيفية انشاء Observable من هذه التقنية.

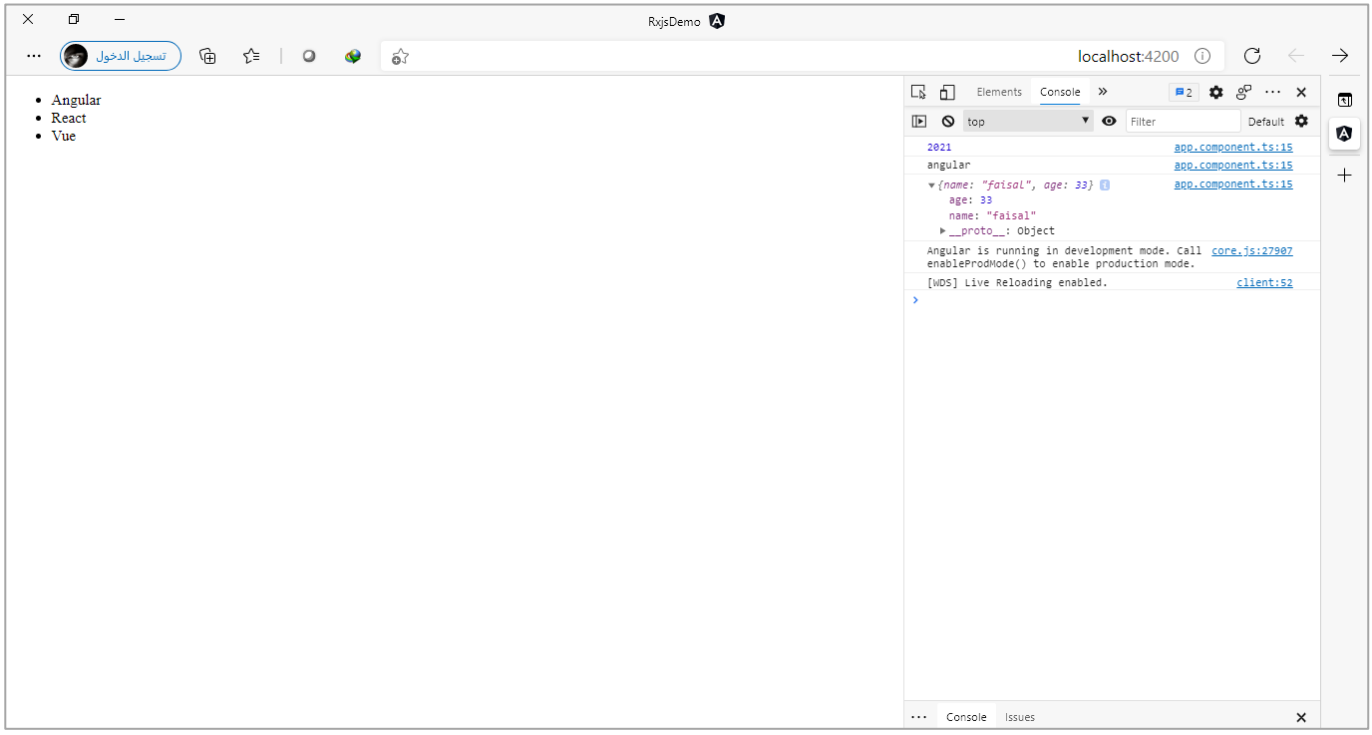
ولكن نستطيع ان نوضح هذه التقنية بشكل مبسط على انها نوعا ما شبيه بالObservable حيث تقوم بعمل emit لمجموعة من القيم بشكل متسلسلة باستخدام الامر yield ومن ثم يتم قراءة هذه القيم باستخدام الدالة next().

ولنعطي مثال لكي تتضح الصورة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  constructor() {}

  ngOnInit(): void {
    from(this.theMeaningOfLife()).subscribe((val) => {
      console.log(val);
    });
  }

  *theMeaningOfLife(): Generator<number | string | { name: string; age: number }> {
    yield 2021;
    yield 'angular';
    yield {
      name: 'faisal',
      age: 33,
    };
  }
}
```

نلاحظ لدينا دالة من النوع Generator تقوم بعمل emit لثلاث أنواع من القيم الأول رقم والثاني نص والثالث عبارة عن كائن Object، ومن ثم قمنا بإنشاء Observable من هذه الدالة عن طريق الدالة from وبنفس الوقت عملنا Subscribe لكي نقرأ هذا Stream من البيانات، والنتيجة في المتصفح سوف تكون كالتالي:



6-1-2- إنشاء Observable من أنواع مختلفة من البيانات بنفس الوقت عن طريق الدالة from:

مع انني أشرت سابقاً اننا لا نستطيع ان نمرر إلا بارامتر واحد فقط لدالة from وهذا معناه اننا سوف ننشأ Observable من نوع واحد من البيانات، إلا اننا نستطيع ان نعمل خدعة معينة تُتيح لنا ان نمرر بارامتر واحد ولكن يحتوي على أكثر من نوع من البيانات، وتتم هذه الخدعة من خلال وضع هذه الأنواع في مصفوفة ومن ثم تمرير هذه المصفوفة كبارامتر لدالة from.

وبما اننا نتعامل مع لغة الجافا سكربت وهي لغة تتميز بانها ديناميكية من حيث تعريف الأنواع، فمن هذه المنطلق نستطيع ان نعرف مصفوفة تحتوي على أنواع متعددة من البيانات وبنفس الوقت نستطيع تحويها جميعاً إلى Observable عن طريق الدالة from والتي بكل تأكيد سوف تعمل لها flatting كما أشرنا سابقاً.

ويجب ملاحظة أنه ليس جميع الأنواع نستطيع ان نمررها كمصفوفة ولكن نستطيع ان نمرر بعض الأنواع مثل النصوص string او الأرقام number او الكائنات objects (مع العلم انها لا تدعم iteration بشكل ضمني)، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  private arr = [[1, 2, 3, 4, 5, 6], 'angular', { name: 'faisal', age: 33 }];
  private obs = from(this.arr);

  constructor() {
    this.obs.subscribe({
      next: (value) => {
        console.log(value);
      },
    });
  }

  ngOnInit(): void {}
}
```

نلاحظ قمنا بتعريف مصفوفة تحتوي على ثلاث أنواع من البيانات النوع الأول هو عبارة عن مصفوفة رقمية والثانية عبارة عن قطعة نصية والأخير عبارة عن كائن، ولنشاهد الآن كيف تكون النتيجة في المتصفح بعدما نمررها إلى الدالة from لكي نقوم بإنشاء Observable من هذه البيانات، كالتالي:

نلاحظ انه تم عمل flatting لهذه المصفوفة الرئيسية وتم تقسيم وقراءة عناصرها كل عنصر على حدا، بدأً من العنصر الأول وهو المصفوفة والعنصر الثاني وهو القطعة النصية وأخيراً العنصر الثالث وهو الكائن.

ومما ذُكر سابقاً نستنتج انه في حال كان لدينا مصفوفة واحدة فقط وأردنا أن نقرئها على شكل Observable مع عمل flat لعناصرها، فلعل هذه الدالة هي المناسبة.

اما في حال أردنا ان نمرر مصفوفة او أكثر من مصفوفة ونقرأها كObservable كما هي بدون أي Flatting فعندئذً لابد ان نستخدم دالة أخرى وهي of وهو ما سوف نتكلم عنه بإذن الله في الجزء التالي.

2-2- of():

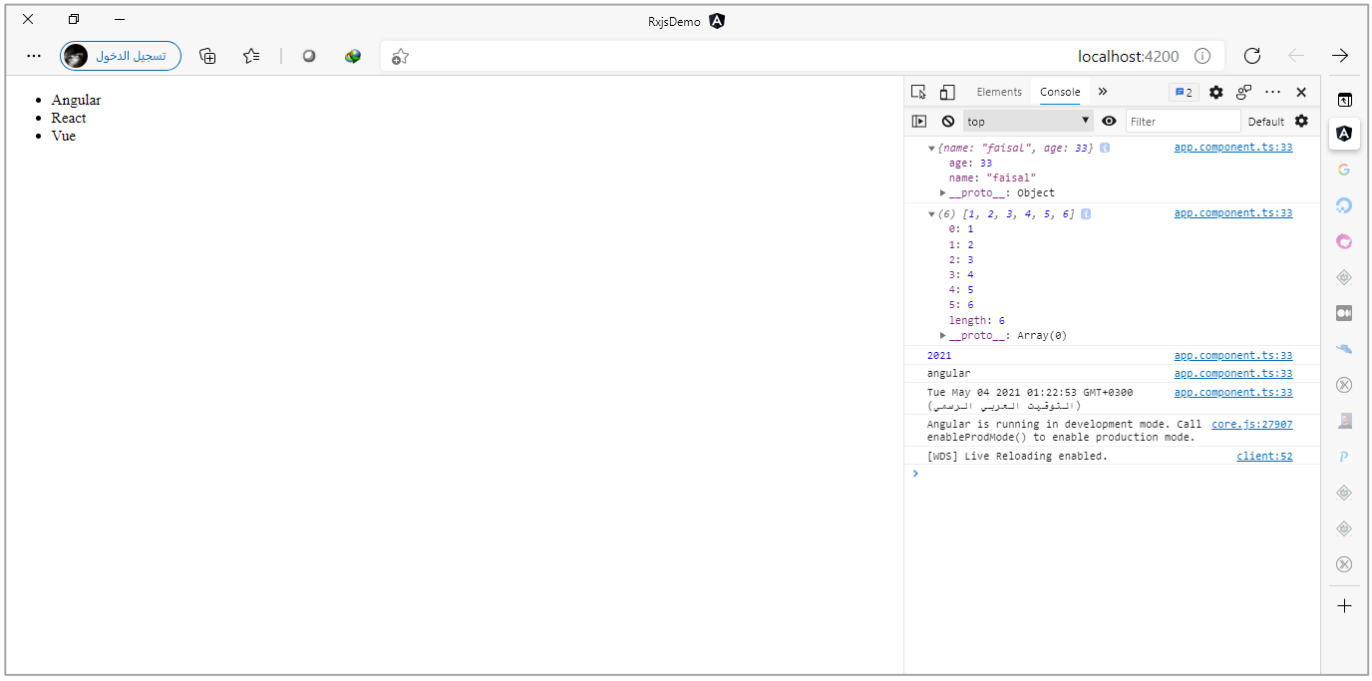
وهذه الدالة تقوم بإنشاء Observable من جميع الأنواع سواء كانت تدعم Iteration بشكل ضمني او لا، ما عدا Promises ودوال Generators فإن هذه الدالة لا تدعمهم ولابد من استخدام الدالة السابقة from، ونستطيع تمرير اي عدد من البارامترات سواء كانت نوع واحد أو أنواع مختلفة، بحيث تُعيد لنا Stream من هذه البارامترات وكل بارامتر على حدا.

ولتوضيح لنعطي المثال التالي الذي يحتوي على بيانات متعددة، ونُريد إنشاء Observable منها جميعاً:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  private obs = of(
    { name: 'faisal', age: 33 },
    [1, 2, 3, 4, 5, 6],
    2021,
    'angular',
    new Date()
  );

  ngOnInit(): void {
    this.obs.subscribe((val) => {
      console.log(val);
    });
  }

}
```

نلاحظ قمنا بتمرير بيانات متعددة على شكل بارامترات لدالة of منها بيانات رقمية ونصية وكائن ومصفوفة وحتى التاريخ والوقت، وسوف تقوم الدالة بإنشاء Observable من هذه البيانات كل نوع على حدا، كالتالي:

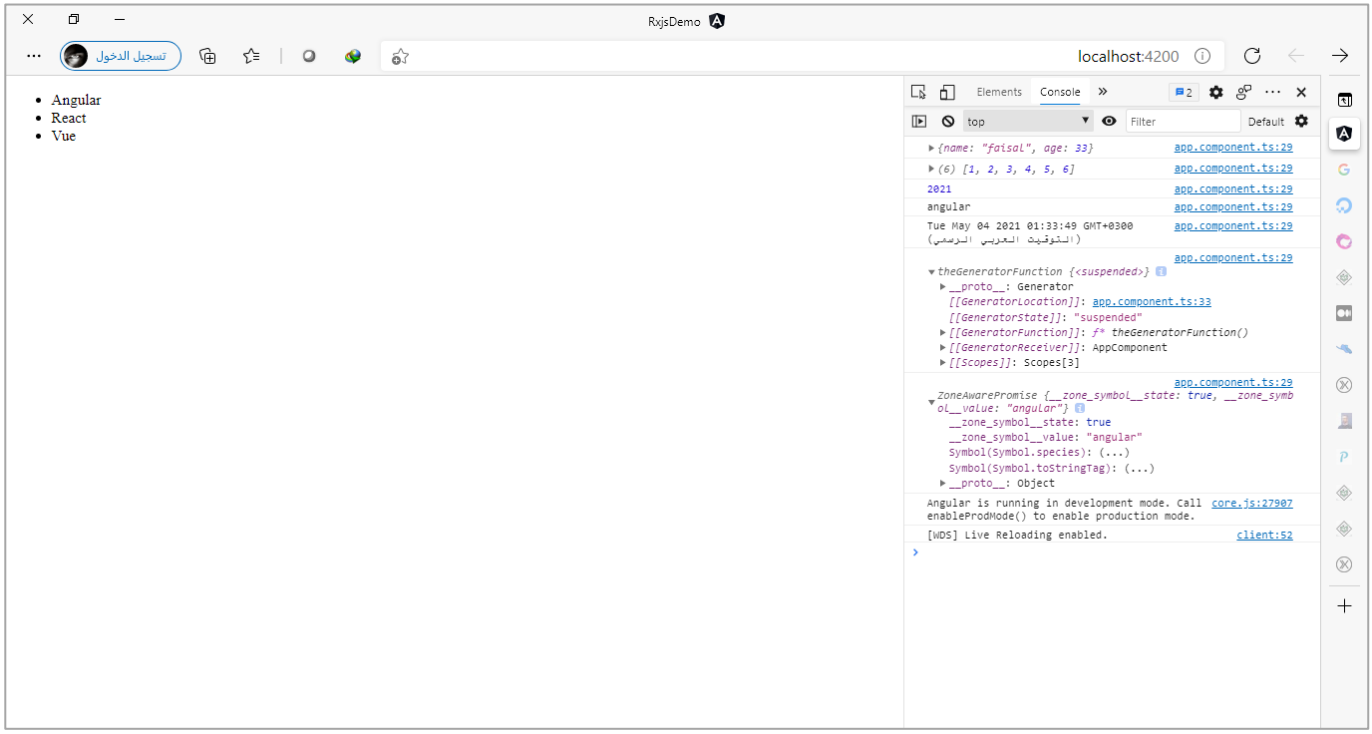


ولكن لا نستطيع ان نمرر Promise او Generator Function إلى الدالة of، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  constructor() {}

  private promise = new Promise((resolve, reject) => {
    resolve('angular');
  });

  private obs = of(
    { name: 'faisal', age: 33 },
    [1, 2, 3, 4, 5, 6],
    2021,
    'angular',
    new Date(),
    this.theGeneratorFunction(),
    this.promise
  );

  ngOnInit(): void {
    this.obs.subscribe((val) => {
      console.log(val);
    });
  }

  *theGeneratorFunction(): Generator<number | string | { name: string; age: number }> {
    yield 2021;
    yield 'angular';
    yield {
      name: 'faisal',
      age: 33,
    };
  }
}
```

والنتيجة تكون، كالتالي:



نلاحظ لم يتم عرض البيانات، لذلك عزيزي المتعلم يجب الانتباه عند التعامل مع هاتين التقنيتين فلابد من استخدام الدالة from معهما عوضاً عن الدالة of.

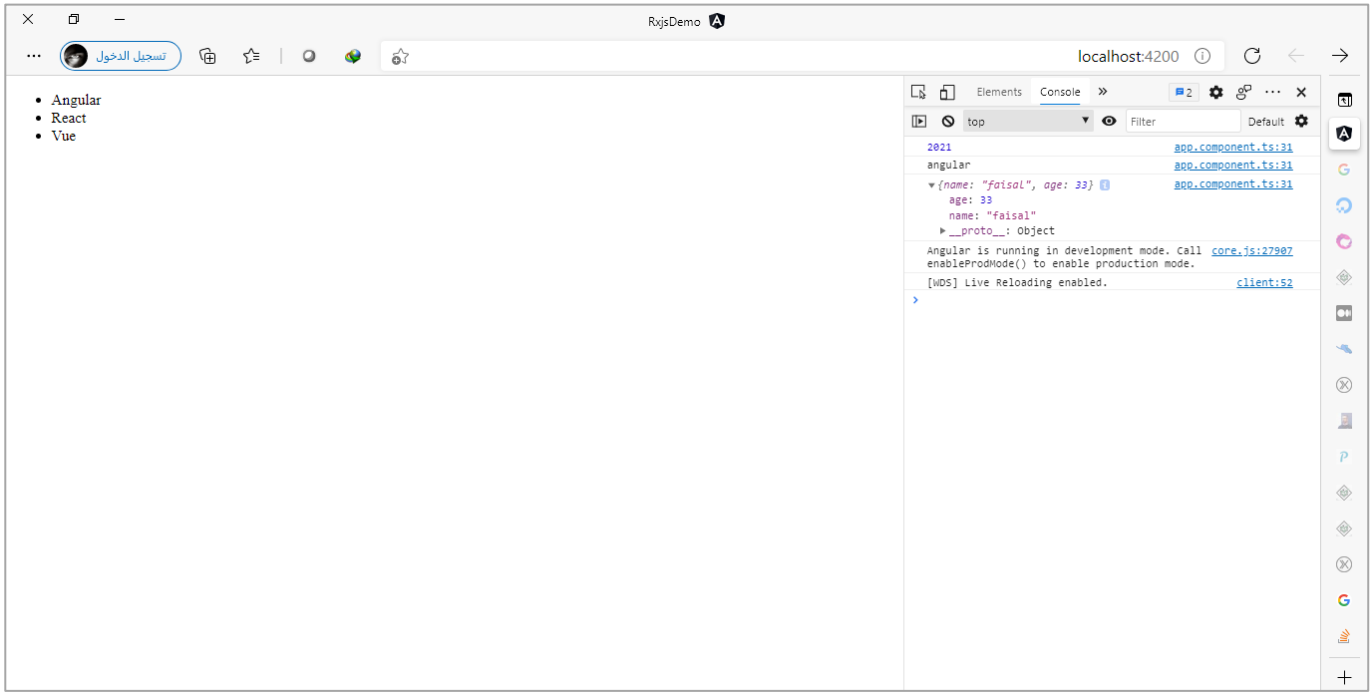
مع العلم ان هنالك خدعة بسيطة نستطيع ان نعملها لكي نجعل الدالة of تعمل مع Generators Function، وتنفع فقط في حال كان لدينا بارامتر واحد فقط، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}
  private obs = of(...this.theGeneratorFunction());

  ngOnInit(): void {
    this.obs.subscribe((val) => {
      console.log(val);
    });
  }

  *theGeneratorFunction(): Generator<number | string | { name: string; age: number }> {
    yield 2021;
    yield 'angular';
    yield {
```

```
      name: 'faisal',
      age: 33,
    };
  }
}
```

اما النتيجة في المتصفح فتكون كالتالي:



وأخيراً أشرنا سابقاً ان الدالة from تقوم بعمل flatting للبيانات بحيث تقوم بعمل emit عنصر ورا الآخر بعكس الدالة of التي تقوم بعمل emit للقيمة كاملة لكل بارامتر، بحيث تعمل flatting لكل بارامتر على حدا ولكن كل بارامتر تقوم بعمل emit للقيمة كاملة، كما شاهدنا في آخر مثال حيث قمنا بتمرير مجموعة من البيانات إلى الدالة of وتم عمل flatting لكل بارامتر على حدا وبنفس الوقت كل بارامتر قامت بعمل emit للقيمة كما هي فلو كانت مصفوفة فسوف تقوم بعمل emit للمصفوفة كاملة وهكذا بقية الأنواع الأخرى.

ولكن نستطيع ان نعمل خدعة معينة بحيث نجعل الدالة of تقوم بعمل flatting لكل القيم لنفس البارامتر وهذه الخدعة تنفع مع المصفوفات والبيانات النصية فقط وفي حال كان هنالك بارامتر واحد فقط.

ولتوضيح لنقوم بتطبيق هذه الخدعة على نفس المثال السابق، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor() {}
  private num = [1, 2, 3, 4, 5, 6];
  private obs = of(...this.num);

  ngOnInit(): void {
    this.obs.subscribe((val) => {
      console.log(val);
    });
  }
}
```

والنتيجة:

2-3- أهم الفروقات بين الدالتين from vs of:

نستطيع حصر أهم أوجه التشابه والفروقات من خلال الجدول التالي:

from	of
تقوم بإنشاء Observable من Promises وأيضاً القيم التي تدعم iteration بشكل ضمني	تقوم بإنشاء Observable من أي نوع من البيانات ماعدا Promises و Generators Function
تستقبل بارامتر واحد اجباري وهو القيمة المُراد إنشاء Observable منها	تستقبل عدد لا محدود من البارامترات (الإصدار السابع والأعلى من مكتبة rxjs اما الإصدارات التي قبلها فتستقبل من 8 إلى 9 بارامترات فقط) بحيث كل بارامتر هو عبارة عن القيمة المُراد انشاء Observable منها.
تدعم Promises و Generators Function	لا تدعم Promises و Generators Function
تقوم بعمل flatting للObservable أي تجزئة القيمة ومن ثم يعمل emit لكل قيمة على حدا	لا تقوم بعمل flatting لقيمة البارامتر نفسه ولكن تعمل flatting لكل بارامتر على حدا

fromEvent()-4-2 :

وهي من الدوال المهمة والتي لها مجالات متنوعة واستخدامات كثيرة، وهذه الدالة بكل بساطة تتعامل مع أحداث Events DOM مثل الضغط على زر الفأرة الأيمن او الأيسر او اجراء عملية النسخ او القص والصق او تمرير شريط التمرير سواء الأفقي او العامودي ...الخ، وللاطلاع على جميع هذه الأحداث الرجاء زيارة الرابط التالي Element - Web APIs | MDN (mozilla.org) مع الانتباه ان هذه الأحداث يتغير مسماها في Angular فمثلاً الحدث onclick سوف يصبح click والحدث oncopy سوف يصبح copy بمعنى نحذف on التي تسبق اسم الحدث.

ومما يميز هذه الدالة انها تتعامل مع أي حدث من أحداث DOM سواء كانت هذه الأحداث تقع على أداة في الصفحة او حتى على الصفحة كاملة ومن ثم تقوم بإنشاء Observable في حال وقوع هذا الحدث.

وتستقبل هذه الدالة بارامتران اللزاميين الأول الأداة التي نُريد ان نراقبها والبارامتر الثاني هو اسم الحدث الذي نُريد ان ننشأ Observable في حال وقوعه.

ونستطيع ان نقول بشكل مبسط ان هذه الدالة تشابه بصورة معينة الدالة المشهورة في Vanilla JavaScript وهي addEventListener، مع الاحتفاظ بالمميزات الرائعة التي تقدمها Observable عند استخدام الدالة fromEvent.

ولتوضيح لنعطي بعض الأمثلة على بعض الأحداث وكيفية انشاء Observable عند وقوع هذه الأحداث، كالتالي:

ملف app.component.html

```
<input #TextBox />
```

اضفنا أداة input وأعطيناها Reference Template باسم TextBox بحيث نراقب الأحداث الواقعة عليها ومن ثم ننشأ Observable منها، كالتالي:

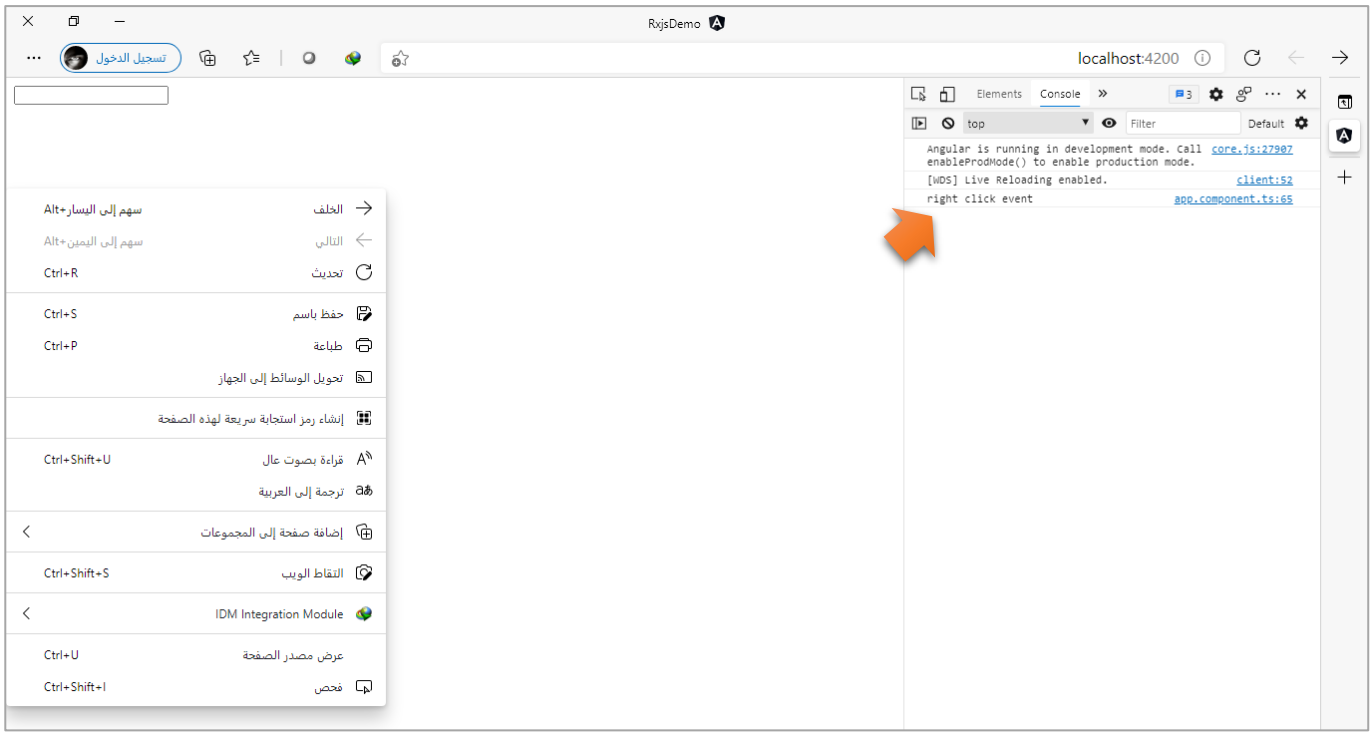
ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  @ViewChild('TextBox', { static: true }) input: ElementRef;

  constructor() {}

  ngOnInit(): void {
    fromEvent(this.input.nativeElement, 'select').subscribe(
```

```
      (value: KeyboardEvent) => {
        console.log('text was selected');
      }
    );
    fromEvent(this.input.nativeElement, 'copy').subscribe(
      (value: KeyboardEvent) => {
        console.log('text was copied');
      }
    );
    fromEvent(document, 'auxclick').subscribe((value: KeyboardEvent) => {
      console.log('right click event');
    });
  }

}
```

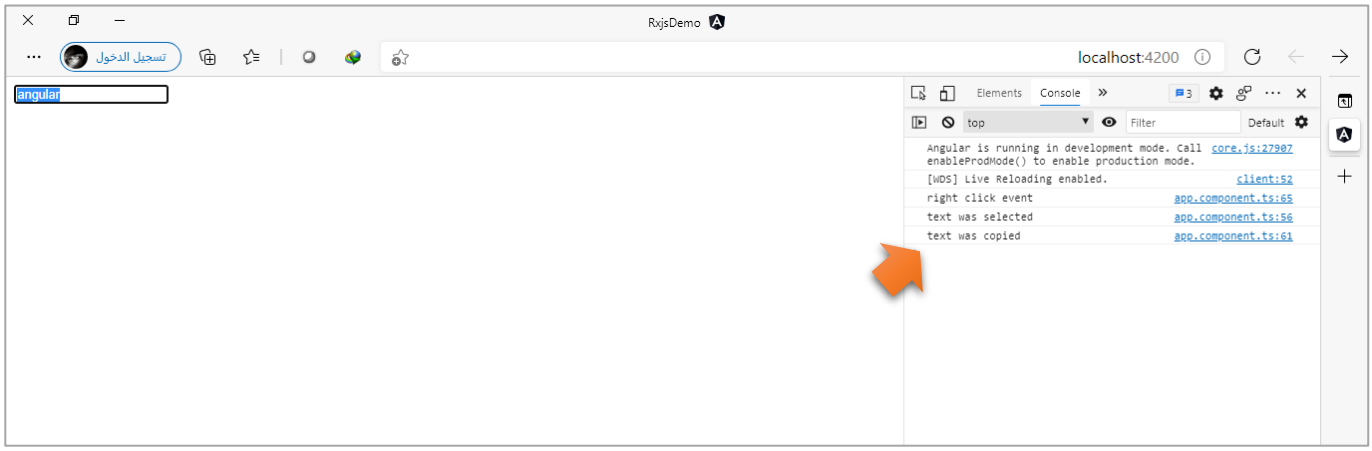
كما نلاحظ قمنا بمراقبة ثلاث أنواع من الأحداث وهم select لكي نراقب في حالة قام المستخدم بتحديد النص في مربع النص، والحدث copy لكي نراقب في حال قام المستخدم بعمل نسخ للنص والحدث الأخير خاص بالصفحة كاملة وهو إذا قام المستخدم بالضغط على زر الفأرة الأيمن في أي منطقة فارغة في الصفحة.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



هنا في حالة الضغط بزر الفأرة الأيمن في أي مكان بالصفحة أي وقوع الحدث auxclick.

وهنا في حالة تحديد النص أي وقوع الحدث select.



وأخيراً في حالة نسخ النص ووقع الحدث copy.

2-5-()firstValueFrom()/lastValueFrom:

وهذه من الدوال الجديدة والتي سوف تحل محل الدالة toPromise، اما من ناحية وظيفتها فتقوم بتحويل Observable إلى Promise حيث الدالة الأولى firstValueFrom تقوم بقراءة اول قيمة يتم عملها emit من هذا Observable ومن ثم يتم تحويلها إلى Promise اما الثاني فتقوم بقراءة آخر قيمة عندما يكتمل هذا Observable، ومن هذا المنطلق تم الغاء الدالة السابقة toPromise حيث اننا سابقاً نستخدم هذه الدالة والتي تُعيد فقط آخر قيمة عندما يكتمل Observable فقط، ولذلك الآن توجد هاتين الدالتين التان تقرئان القيمة الأولى والأخيرة.

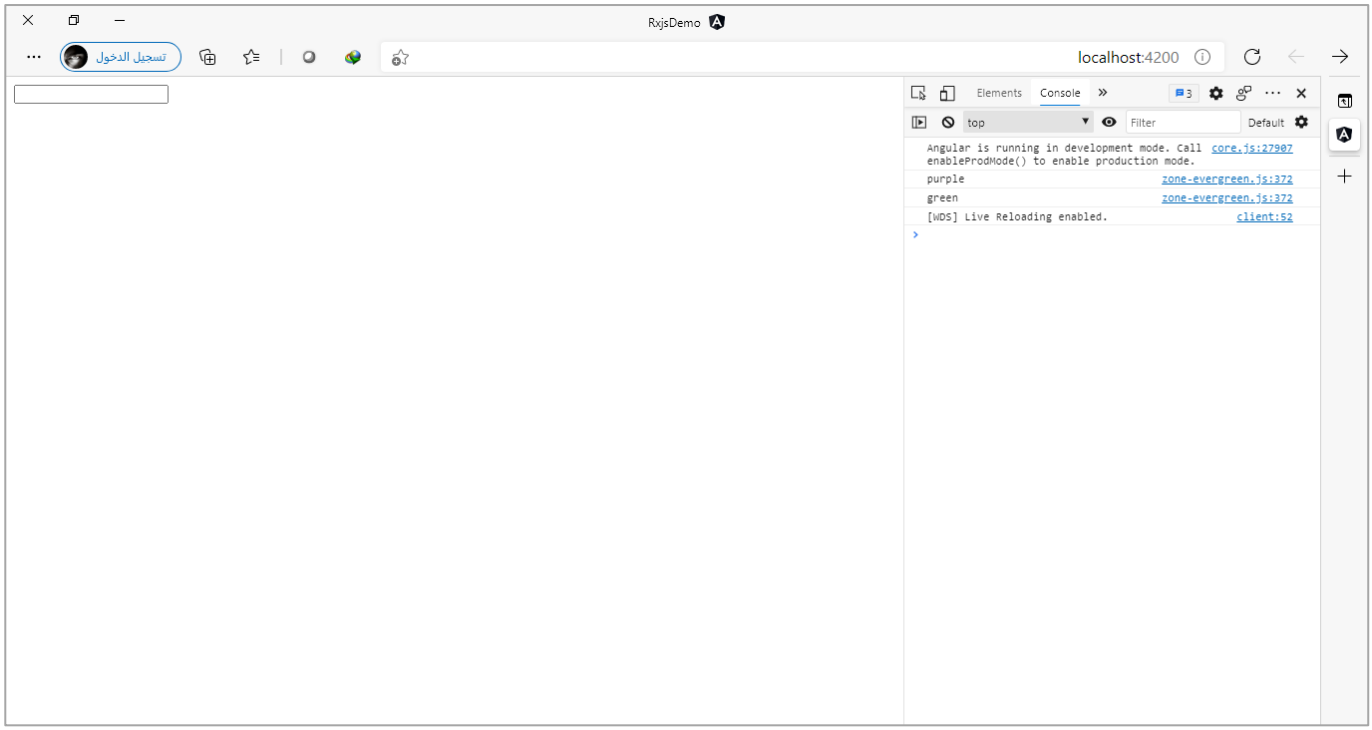
ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { firstValueFrom, lastValueFrom, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    const obs = of('purple', 300, 500, true, 'green');
    firstValueFrom(obs).then(console.log);
    lastValueFrom(obs).then(console.log);
  }

}
```

نلاحظ قمنا بإنشاء Observable باستخدام الدالة of ومررنا مجموعة من القيم ومن ثم استخدمنا الدالتين السابقتين لقراءة اول قيمة وآخر قيمة بعد تحويلهما إلى Promise، وسوف تكون النتيجة كالتالي:



وبطبيعة الحال في حال وجود خطأ نستخدم catch مع then (اساسيات Promises وليس هنا المقام لشرحها).

EMPTY()/EmptyError-6-2:

الدالة EMPTY تقوم بإعادة Observable فارغ وتغلق هذا Observable، وقد نحتاج في بعض الأحيان ان نغلق Observable او مثلاً نُريد ان نتأكد من البيانات القادمة من قاعدة البيانات فإذا كان هنالك بيانات او قم بإعادة Observable فارغ، اما EmptyError فهي في الحقيقة ليست دالة وانما كلاس نستطيع الاستفادة منه لنعرف نوع الخطأ القادم لنا من Observable فإذا كان من هذا النوع فعندها نستطيع ان نظهر مثلاً رسالة معينة للمستخدم او نقوم بإي أمر آخر.

ولتوضيح لنقوم بتعديل المثال السابق ونضيف له الدالة EMPTY، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { lastValueFrom } from 'rxjs';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    lastValueFrom(EMPTY).catch((err) => {
      if (err instanceof EmptyError) console.log(err.message);
    });
  }
}
```

2-7-()defer:

وهذه من الدوال قليلة الاستخدام وايضاً استخدام هذه الدالة محدود، ولكنها حقيقة مهمة لذلك ارتأيت ان اضيفها في هذا الكتاب، وتقوم وضيفتها الأساسية بعمل تغليف لجزء معين من الكود بحيث لا يقرأه المتصفح او بصيغة اصح المترجم compiler الخاص بالغة الجافا سكريبت إلا عندما نقوم بعمل subscribe لهذه الدالة، ولتوضيح لنعطي المثال التالي، (هذا المثال تم أخذه من دورة rxjs من قناة angular army).

لنفرض انه لدينا زر بحيث كلما تم الضغط على هذا الزر يتم إعطاء قيمة عشوائية جديدة، وهنالك طرق كثيرة للقيام بهذا الأمر منها عن طريق استخدام @ViewChild والدالة fromEvent، كالتالي:

ملف app.component.html

```
<button #button>Click Me !</button>
```

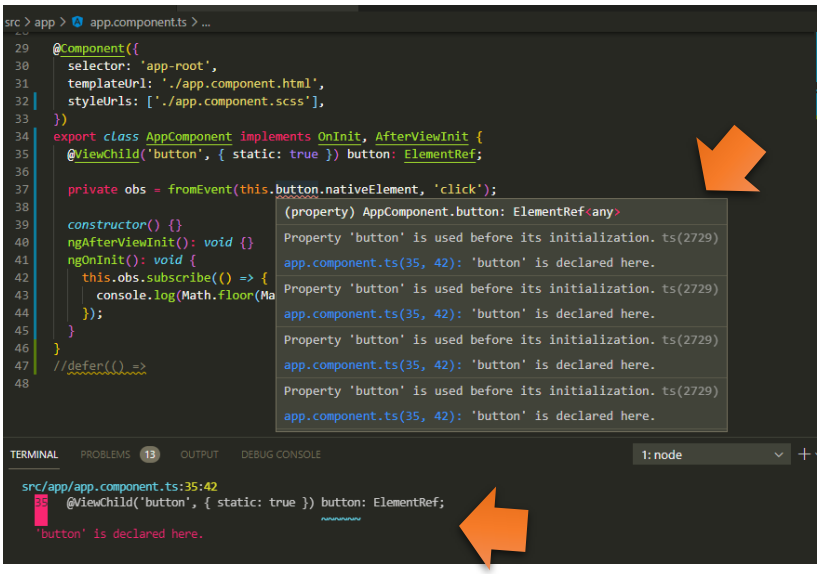
وفي ملف class لهذا component نضيف logic، التالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { defer, fromEvent } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit, AfterViewInit {
```

```
  @ViewChild('button', { static: true }) button: ElementRef;

  private obs = fromEvent(this.button.nativeElement, 'click');

  constructor() {}
  ngOnInit(): void {
    this.obs.subscribe(() => {
      console.log(Math.floor(Math.random() * 10));
    });
  }
}
```

الكود السابق لن يعمل وسوف تظهر رسالة، خطأ كما في الصورة التالية:



ومعنى رسالة الخطأ انه تم استخدام هذه المتغير الذي اسميته button قبل ان نعمل له تهيئة، وهنالك حلول متعددة منها ان نقوم بتغليف السطر البرمجي باستخدام الدالة defer بحيث يصبح المتغير obs يُشير إلى الدالة defer بدلاً من الدالة fromEvent، وبهذه الطريقة يصبح هذا السطر البرمجي غير مرئي للcompiler إلى ان يتم عمل subscribe له والمتمثل في مثالنا هذا بالمتغير obs، اما الآن لنقوم بتطبيق ما قلناه سابقاً بحيث يصبح الكود كالتالي:

app.component.ts ملف

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { defer, fromEvent } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit, AfterViewInit {

  @ViewChild('button', { static: true }) button: ElementRef;

  private obs = defer(() => fromEvent(this.button.nativeElement, 'click'));
```

```
  constructor() {}
  ngOnInit(): void {
    this.obs.subscribe(() => {
      console.log(Math.floor(Math.random() * 10));
    });
  }

}
```

اما النتيجة في المتصفح، فتكون كالتالي:

نلاحظ مع كُل ضغطة على الزر يتم قراءة هذا event عن طريق fromEvent بدون أي مشاكل ومن ثم يتم تنفيذ الكود وهو انشاء قيم عشوائية ومتغيرة.

2-8- interval()/timer():

وهذه الدوال للتعامل مع الوقت وبنفس الوقت تقومان بإنشاء Stream of Data (Observable)، حيث تقوم الدالة الأولى باستقبال بارامتر واحد بالملي ثانية – 1000 ملي ثانية تُساوي ثانية واحدة – وهذا معناه بعد مرور الوقت المُعطى قم بتشغيل العداد بحيث يبدأ من الصفر، ولتوضيح لنعطي المثال التالي:

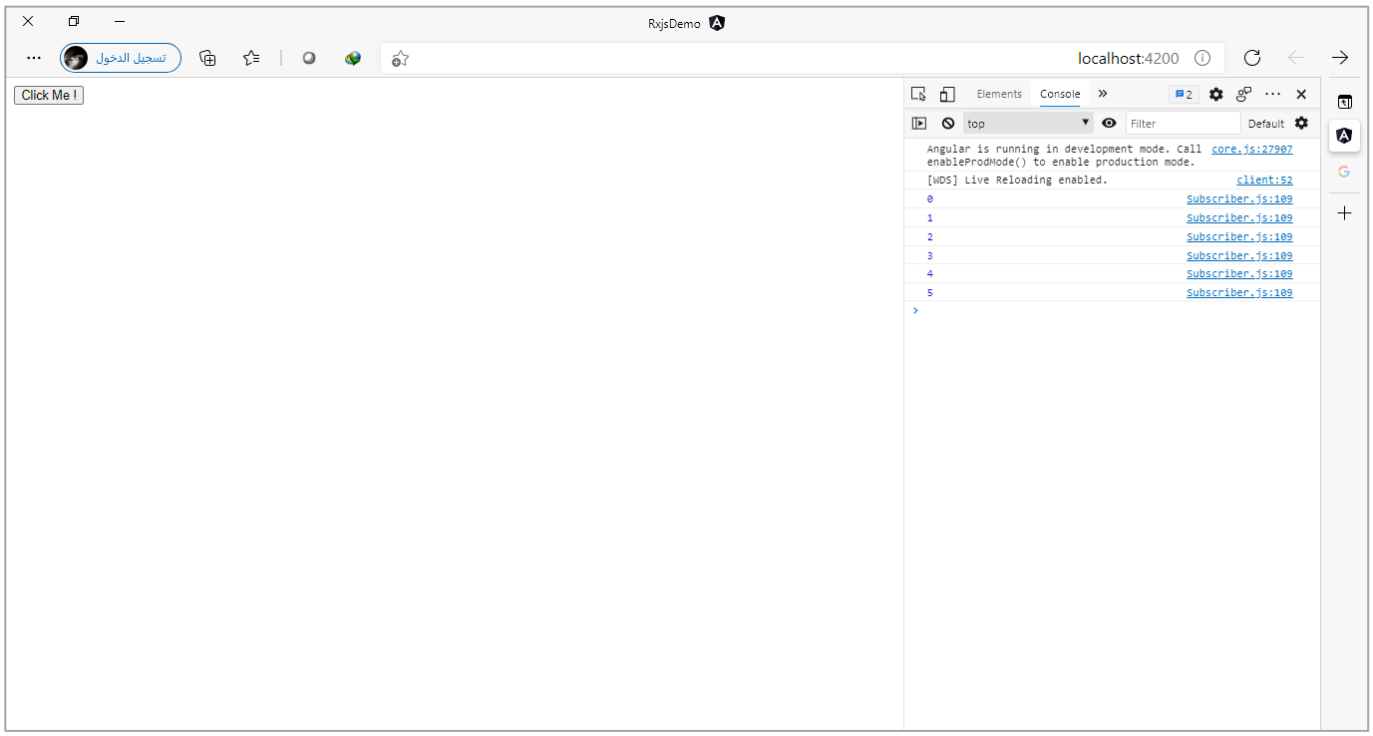
ملف app.component.ts

```
import { Component, ElementRef, OnInit ViewChild } from '@angular/core';
import { interval } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private time = interval(2000);
```

```
  constructor() {}
  ngOnInit(): void {
    this.time.subscribe(console.log);
  }
}
```

المثال السابق كما هو واضح نقول له كل ثانيتين – 2000 ملي ثانية – قم بعمل emit لقيمة رقمية بحيث هذه القيم تبدأ من الصفر وتكون على شكل عداد تصاعدي.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



هذا من ناحية الدالة الأولى اما الدالة الثانية timer فتختلف قليلاً حيث تستقبل بارامترين الأول هو متى يبدأ العداد والثاني الوقت المستقطع بين كل قيمة والتي تليها، ولتوضيح لنقوم بتعديل المثال السابق لكي يتوافق مع هذه الدالة، كالتالي:

app.component.ts ملف

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { timer } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit, AfterViewInit {
  private time = timer(5000, 2000);

  constructor() {}
```

```
  ngOnInit(): void {
    this.time.subscribe(console.log);
  }
}
```

كما هو واضح استخدمنا الدالة timer ومررنا لها خمس ثواني وثانيتين، وهذا معناه كأننا نقول بعد خمس ثواني قم ببدء العداد – يعمل emit للقيم على شكل Observable – ومن ثم كل ثانيتين قم بأرسال قيمة أي يُرسل القيمة الأولى ومن ثم ينتظر ثانيتين ويُرسل القيمة الثانية وهكذا بقية القيم.

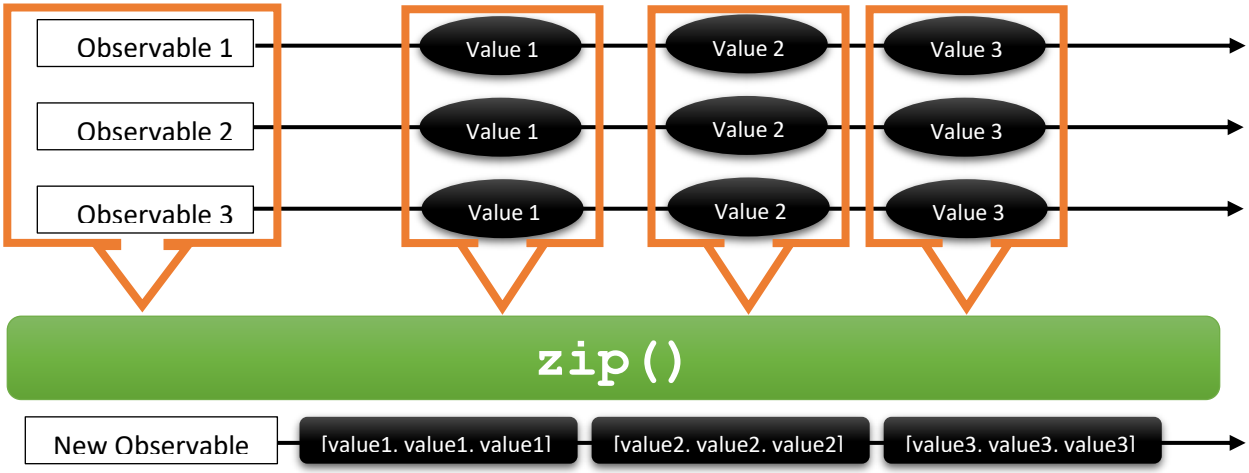
اما الآن لنشاهد النتيجة في المتصفح، كالتالي:

وأخيراً يجب الاشارة إلى انه في كلا الدالتين فإن Observable لن يتوقف، وسوف يستمر في بعمل emit ولابد ان نقوم بإيقافه عن طريق استخدام بعض الدوال مثل الدالة take، والتي سوف نتكلم عنها بإذن الله عندما نصل إلى القسم الثاني من دوال rxjs.

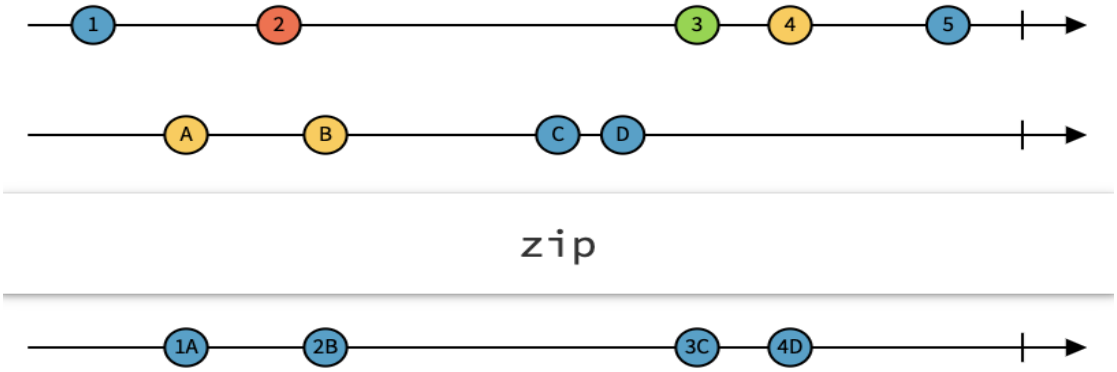
2-9- zip():

وتقوم هذه الدالة باستقبال أكثر من Observable ومن ثم تقوم بدمجها بحيث تصبح Observable واحد، اما من ناحية طريقة دمج هذه Observables المختلفة فتقوم بدمج كل قيمة من كل Observable سوياً في مصفوفة لوحده ومن ثم تنتقل إلى القيم التالية وتقوم ايضاً بدمجها سويا وهكذا إلى ان تنهي من جميع القيم لجميع الObservables المختلفة، ولكن يجب الانتباه إلى مجموعة من النقاط، كالتالي:

- لن تعمل هذه الدالة إلا بعدما يتم عمل emit لقيمة واحدة على الأقل من جميع Observables المختلفة.
- تقوم هذه الدالة بعمل جمع كل قيمة مع القيمة المقابلة لها على شكل مصفوفة، كما في الشكل التوضيحي التالي:



بطبيعة الحال الشكل السابق هو شكل توضيحي، وليس بالضرورة ان يتم عمل emit لكل القيم من جميع Observables الأخرى بنفس الوقت ولكن في حال تأخر أي قيمة فإن الدالة zip سوف تنتظر إلى ان تكتمل جميع القيم المقابلة لهذه القيمة من جميع Observables المختلفة ومن ثم تقوم بجمعها على شكل مصفوفة، وهو ما اشرت إليه في النقطة السابقة (لن تعمل هذه الدالة إلا بعدما...الخ)، ولعل الشكل التوضيحي التالي يوضح المقصود بشكل أفضل:



- من الشكل السابق نلاحظ انه في حال لم يكن هنالك أي قيمة مقابلة في Observable معين في Observables الأخرى فإن الدالة zip سوف تتجاهل هذه القيمة، كما هو واضح في القيمة 5 في الشكل السابق.
- ويجب الانتباه انه في حال اكتمال جميع القيم من جميع Observables فإن هذه الدالة سوف تقوم بعمل complete لي Observable الجديد.
- ومن الأمور المهمة التي يجب الانتباه لها وهو في حالة ان أحد Observables الداخلة إلى هذا Observable تم عمل له completed لأي سبب من الأسباب بدون لا يقوم بعمل emit لأي قيمة فإن الدالة zip سوف تتجاهل جميع القيم وتعمل complete مع انها لم تعمل دمج لأي قيمة، بمعنى آخر الدالة next في Subscriber لن يتم تفعيلها.
- وأخيراً في حالة وجود أي خطأ في أحد Observables فإن هذه الدالة سوف تتجاهل جميع القيم وتقوم بإلقاء Exception، لذلك لابد ان نقوم بالتعامل مع الخطأ من خلال جالة error الموجودة في subscriber، كما شرحنا في الفصل السابق من هذا الكتاب.

اما الآن لنقوم بإعطاء مثال لتوضيح ما قيل سابقاً، كالتالي:

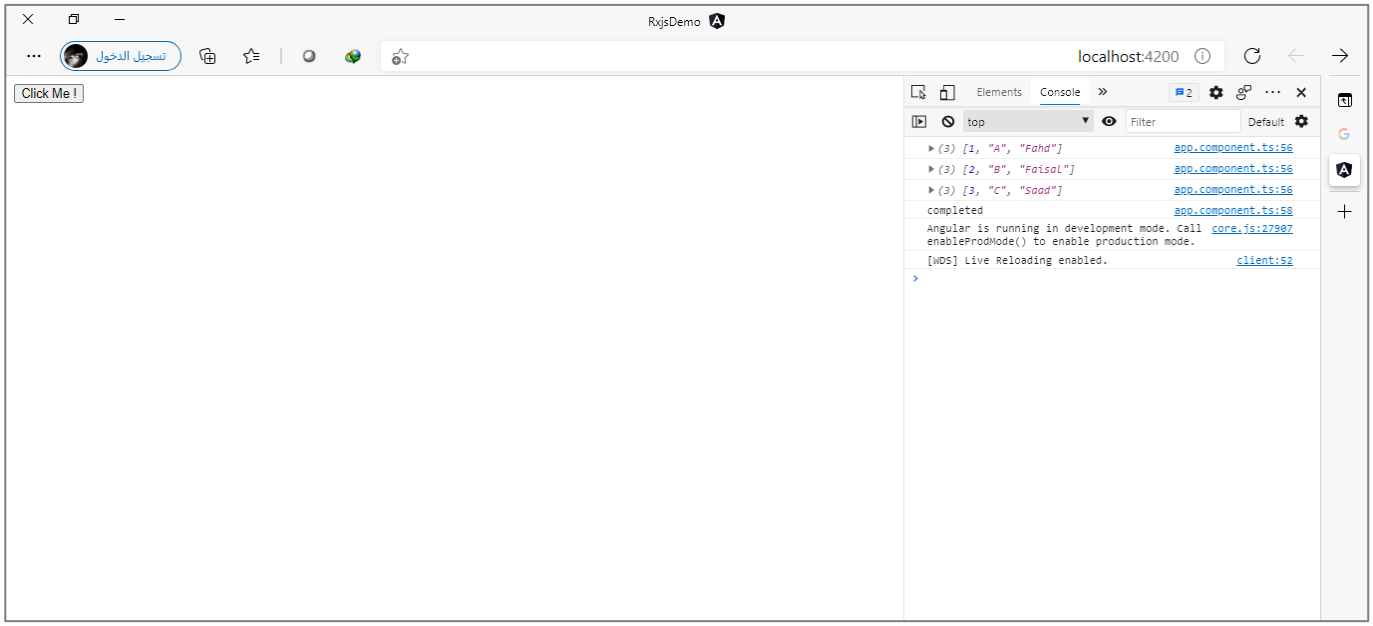
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, zip } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit, AfterViewInit {

  private obs1 = of(1, 2, 3, 4, 5);
  private obs2 = of('A', 'B', 'C', 'D');
  private obs3 = of('Fahd', 'Faisal', 'Saad');

  ngOnInit(): void {
    zip(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.error(error),
      complete: () => console.log('completed'),
    });
  }

}
```

هنا نتعامل مع القيم في حال عمل لها emit من خلال الدالة next في subscriber

هنا نتعامل مع الخطأ في حال وجوده

كما هو واضح من خلال الشفرة السابقة يوجد ثلاثة Observables باسم (obs1-obs2-obs3) ومن ثم قمنا بدمج هذه الـObservables باستخدام الدالة zip ومن ثم قمنا بعمل subscribe وذلك لكي نتعامل مع الحالات الثلاثة لأي Observable وهي اما النجاح next او الخطأ error او اكتمال البيانات complete (هذه المفاهيم تم شرحها في الفصل السابق من هذا الكتاب)، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ انه أخذ القيم المُقابِلة في كل Observable وتجاهل أي قيمة ليس لها مقابل في Observables الأخرى.

اما الآن لنقوم بتأخير احدى الObservables لكي نُحاكي ان بعض القيم قد تتأخر بعمل emit لها لكي نفهم كيف تتعامل دالة zip مع هذا النوع من الحالات، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, zip } from 'rxjs';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private obs1 = of(1, 2, 3, 4, 5).pipe(delay(3000));
  private obs2 = of('A', 'B', 'C', 'D');
  private obs3 = of('Fahd', 'Faisal', 'Saad');

  constructor() {}
  ngOnInit(): void {
    zip(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.error(error),
      complete: () => console.log('completed'),
    });
  }
}
```

لا تهتم عزيزي المتعلم بالأمر البرمجي الذي اشرت عليه بالسهم في الأعلى ولا تُحاول فهمه حالياً، لأننا سوف نشرحه بشكل مُستفيض بإذن الله في القسم الثاني من دوال rxjs، وما يهمنا فقط ان نعلم ان هذا الأمر يقوم بتأخير تنفيذ هذا الأمر لمدة ثلاث ثواني، والسبب في ذلك لأننا نريد ان نعرف ماذا تفعل دالة zip في حال تأخر أحدى الدوال بعمل emit لقيمة معينة، والنتيجة سوف تكون انها سوف تنتظر ثلاث ثواني إلى ان يتم عمل emit للقيم في Observable الذي اسميناه obs1 ومن ثم تقوم بدمج هذه القيم مع القيم المقابلة لها في Observables الأخرى.

حسناً الآن لنقوم بتجربة حالة أخرى وهي في حال تم عمل emit لقيمة واحدة من أحد Observables ماذا سوف تفعل هذه الدالة في هذا الوضع؟ كما في الشفرة التالية:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, zip } from 'rxjs';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit, AfterViewInit {

  private obs1 = of(1, 2, 3, 4, 5).pipe(delay(3000));
  private obs2 = of('A', 'B', 'C', 'D');
  private obs3 = of('Fahd');

  constructor() {}
  ngOnInit(): void {
    zip(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.error(error),
      complete: () => console.log('completed'),
    });
  }
}
```

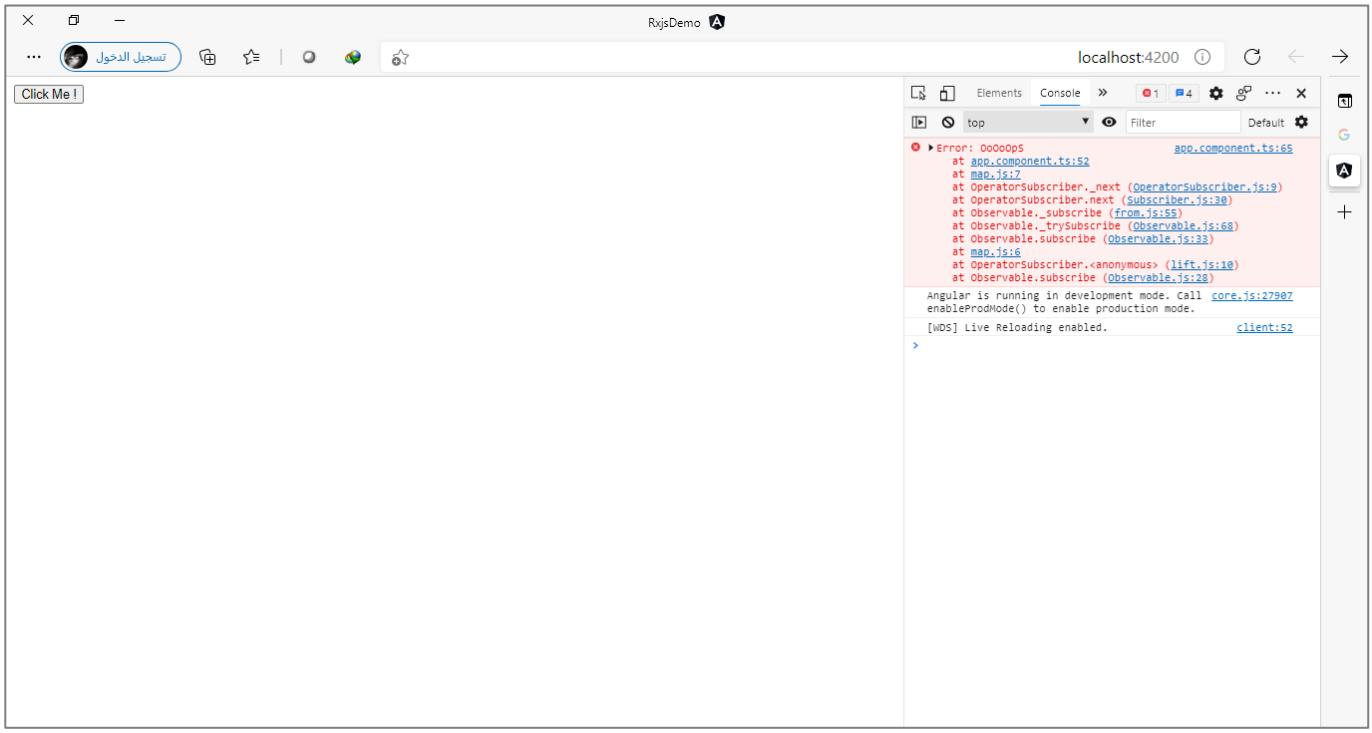
نلاحظ ان Observable ذو الاسم obs3 قام بعمل emit لقيمة واحده فقط ومن ثم اغلق نفسه بعمل complete، لذلك الدالة zip سوف تقوم بعمل تجاهل لجميع القيم التي ليس لها مقابل في Observables الأخرى، بمعنى آخر انها سوف تأخذ القيمة 1 في obs1 والقيمة التي تُقابلها في obs2 وهي A واخيراً القيمة التي تقابل هذه القيم في obs3 وهي Fahd، ولكن سوف تتجاهل باقي القيم لأن القيمة 2 في obs1 والقيمة B في obs2 ليس لها قيمة مُقابِلة في obs3 لذلك سوف يتم تجاهلها وهكذا بقية القيم الأخرى، وسوف تكون النتيجة، كالتالي:

وهنالك حالة أخرى أريد استعراضها مع هذه الدالة وهي في حال وقوع خطأ معين في أحد Observables، فكيف تتعامل هذه الدالة معه؟ كما في الشفرة التالية:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, zip } from 'rxjs';
import { map, delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private obs1 = of(1, 2, 3, 4, 5).pipe(delay(3000));
  private obs2 = of('A', 'B', 'C', 'D').pipe(
    map((value) => {
      if (value === 'D') {
        throw new Error('OoOoOpS');
      } else {
        return value;
      }
    })
  );
  private obs3 = of('Fahd', 'Faisal', 'Saad');

  constructor() {}

  ngOnInit(): void {
    zip(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.error(error),
      complete: () => console.log('completed'),
    });
  }
}
```

ايضاً عزيزي المتعلم لا تهتم حالياً بالأسطر البرمجية التي قمت بالتأشير عليها فسوف نشرحها بالتفصيل لاحقاً بإذن الله، اما ما يهمنا حالياً ان هذه الأسطر تقوم بإلقاء Exception معين في حال كانت أحدى قيم obs2 هي D، ولو تلاحظ عزيزي المتعلم انني وضعت الشرط لإلقاء هذا الخطأ إذا كانت القيمة هي D مع العلم ان هذه القيمة ليس لها قيم مقابلة في obs3 وهذا يعني ان هذه القيمة سوف يتم تجاهلها من خلال الدالة zip، ولكن في حال وجود أي خطأ سوف تقوم هذه الدالة بتجاهل جميع القيم من جميع Observables الأخرى، ومن ثم تقوم بإلقاء خطأ، والذي قمنا بتعامل معه وعرضه للمستخدم في الدالة error في Subscriber، اما النتيجة فتكون كالتالي:

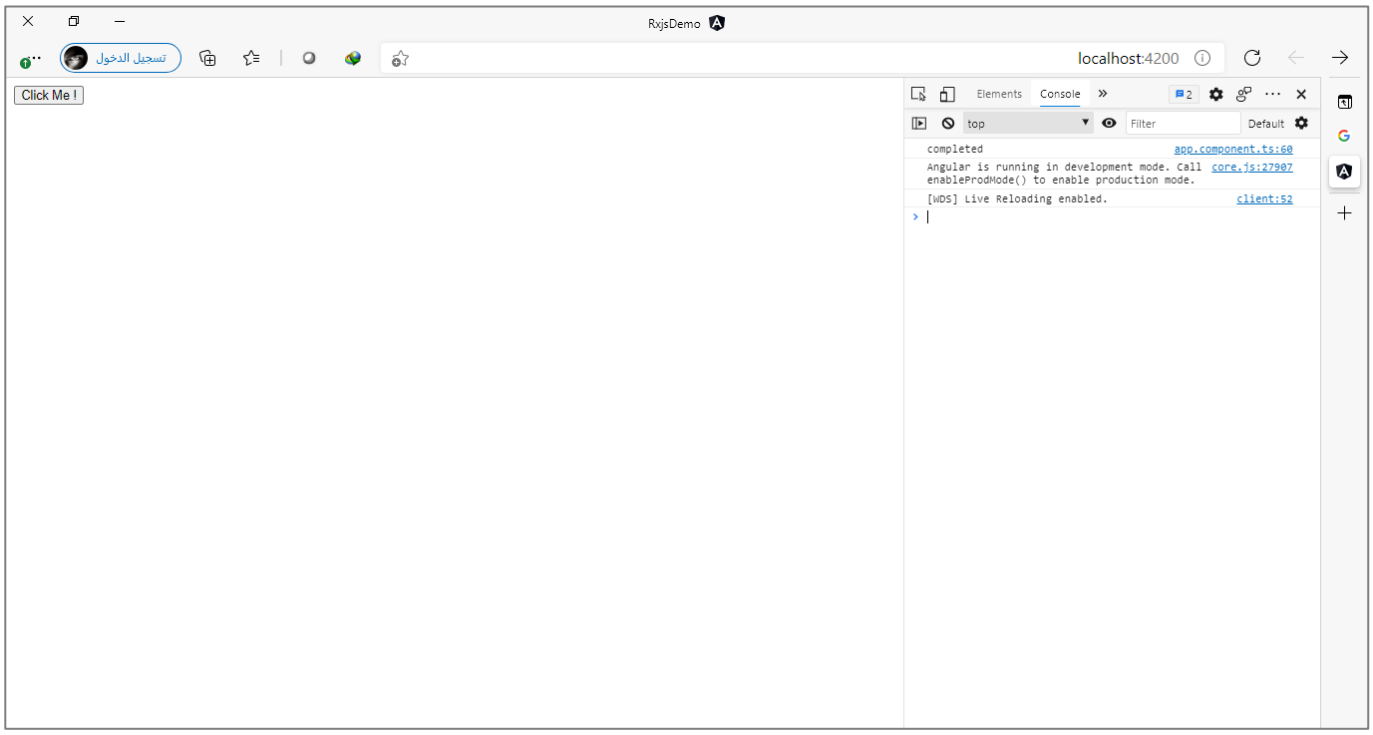


وهنالك حالة أخرى أريد ان استعرضها معكم وهي في حالة ان أحد Observables الأخرى تم عمل لها completed بدون ان تعمل أي emit لأي قيمة، ففي هذه الحالة سوف يتم تجاهل جميع القيم وتجاهل الدالة next في subscriber وتفعيل الدالة complete، كما في الشفرة البرمجية التالية:

ملف app.component.ts

```
import { Component, OnInit} from '@angular/core';
import { of, zip, Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit, AfterViewInit {

  private obs1 = of(1, 2, 3, 4, 5);
  private obs2 = new Observable((observer) => observer.complete());
  private obs3 = of('Fahd', 'Faisal', 'Saad');

  constructor() {}
  ngOnInit(): void {
    zip(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.error(error),
      complete: () => console.log('completed'),
    });
  }
}
```

نلاحظ أن obs2 جعلناه يعمل completed بدون ان يعمل emit لأي قيمة، لذلك سوف تقوم الدالة zip بعمل تجاهل لجميع القيم ولن تُفعل الدالة next في subscriber، بحيث تكون النتيجة كالتالي:



ونفس الوضع في حال كانت إحدى Observables تُعيد EMPTY (أحدى الدوال التي تكلمنا عنها سابقاً) فإن الدالة zip سوف تعمل complete مع تجاهل جميع القيم (لو قمنا بتطبيقها على المثال السابق فإن السطر البرمجي سوف يكون (obs2=EMPTY)، ويجب الانتباه ملاحظة ان نفس Observable لا يوجد به أي قيمة، وليس انه يعمل emit لقيم فارغة ففي هذه الحالة سوف سيتم قراءة القيم الفارغة التي تم عملها emit وتعرضها الدالة zip كقيم فارغة.

وآخر حالة أُريد ان نستعرضها سوياً مع هذه الدالة هي في حالة أردنا نعمل Destructuring للArray التي تنتج من Observables، كما في المثال التالي:

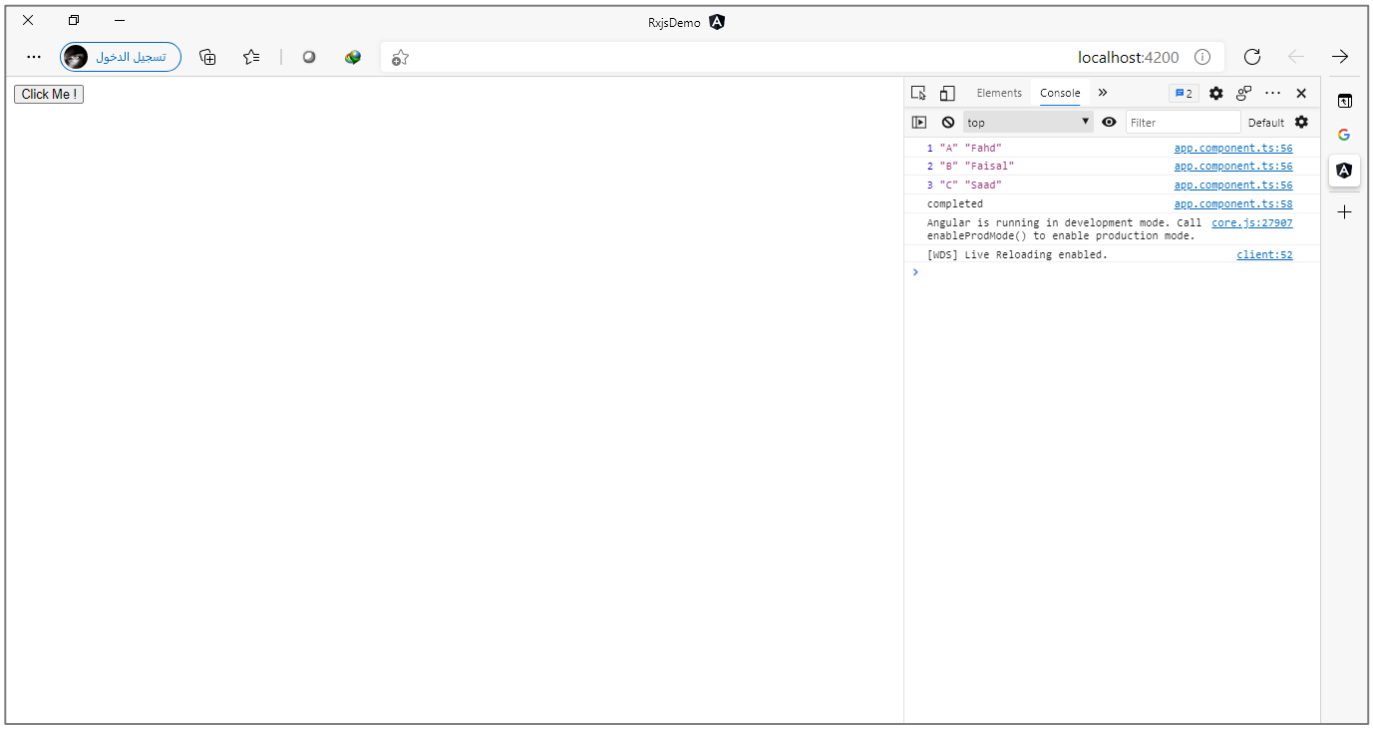
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, zip } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private obs1 = of(1, 2, 3, 4, 5);
  private obs2 = of('A', 'B', 'C', 'D');
  private obs3 = of('Fahd', 'Faisal', 'Saad');
```

```
  constructor() {}

  ngOnInit(): void {
    zip(this.obs1, this.obs2, this.obs3).subscribe({
      next: ([o1, o2, o3]) => console.log(o1, o2, o3),
      error: (error) => console.error(error),
      complete: () => console.log('completed'),
    });
  }

}
```

حيث أن o1 هو متغير (لك حرية اختيار الاسم الذي تُريده) يمثل اول Observable تم تمريره لدالة zip والذي هو في حالتنا هذه هو obs1، وهكذا بقية المتغيرات على التوالي.

اما النتيجة في المتصفح فتكون كالتالي:



10-2- combineLatest():

وهذه الدالة هي ايضاً من الدوال التي تقوم بعمل دمج لمجموعة من Observables في Observable واحد، وهي بذلك مشابهه إلى حد كبير بالدالة السابقة zip من عدة أوجه:

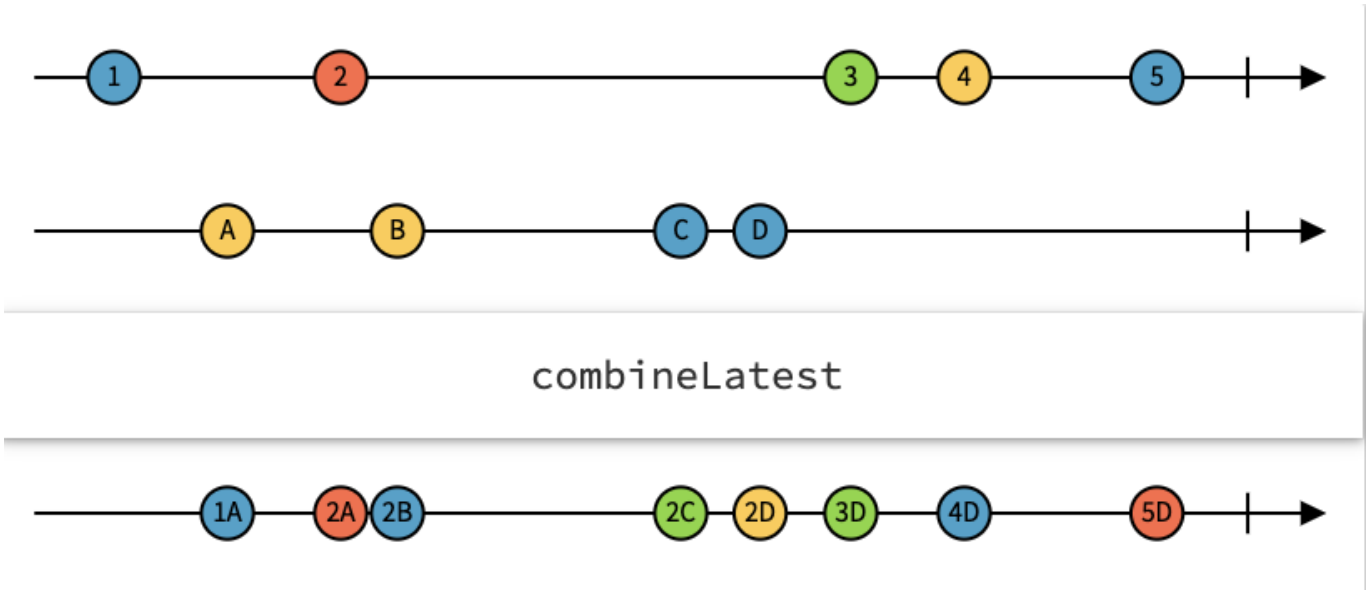
- لن تعمل هذه الدالة إلا بعدما يتم عمل emit لقيمة واحدة على الأقل من جميع Observables المختلفة.
- ويجب الانتباه انه في حال اكتمال جميع القيم من جميع Observables فإن هذه الدالة سوف تقوم بعمل complete للObservable الجديد.
- ومن الأمور المهمة التي يجب الانتباه لها وهو في حالة ان أحد Observables الداخلة إلى هذا Observable تم عمل له completed لأي سبب من الأسباب بدون لا يقوم بعمل emit لأي قيمة فإن هذه الدالة سوف تتجاهل

جميع القيم وتعمل complete مع انها لم تعمل دمج لأي قيمة، بمعنى آخر الدالة next في Subscriber لن بتم تفعيلها.

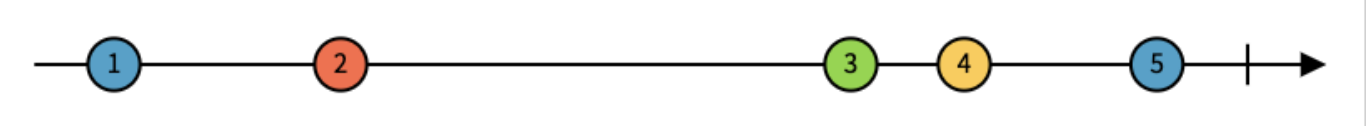
- وأخيراً في حالة وجود أي خطأ في أحد Observables فإن هذه الدالة سوف تتجاهل جميع القيم وتقوم بإلقاء Exception، لذلك لابد ان نقوم بالتعامل مع الخطأ من خلال الدالة error الموجودة في subscriber، كما شرحنا في الفصل السابق من هذا الكتاب.

وتختلف عنها في النقاط التالية:

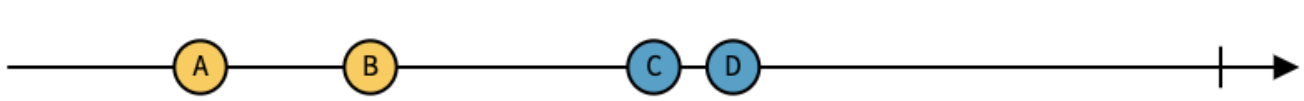
- يتم تمرير Observables المختلفة إلى الدالة combineLatest اما على شكل مصفوفة او على شكل كائن بعكس الدالة zip التي ممكن ان نمرر Observables لها اما مباشرة أو على شكل مصفوفة.
- ولعل الفرق الجوهري هو في طريقة دمج القيم في كل Observable في Observable واحد، حيث عن طريق هذه الدالة لا نقوم بدمج كل قيمة والقيمة المقابلة لها مع انتظار القيمة في حال تأخرها، وانما هنا يتم انتظار القيم الأولى فقط من كل Observable ومن ثم في حال قام أحد Observable الأخرى بعمل emit لقيمة جديدة فإن هذه الدالة سوف تأخذ آخر قيمة موجودة في Observables الأخرى ومن ثم تدمُجها مع هذه القيمة الجديدة التي تم عمل لها emit آنفاً. ولتوضيح لنشاهد الشكل التوضيحي التالي ومن ثم نقوم بشرحه لتتضح الصورة بشكل أفضل:



ومن خلال الشكل السابق سوف نستنتج انه لدينا اثنين Observables الأول قام بعمل emit لمجموعة من القيم، كالتالي:



اما Observable الثاني فأيضاً قام بعمل emit لمجموعة من القيم، كالتالي:



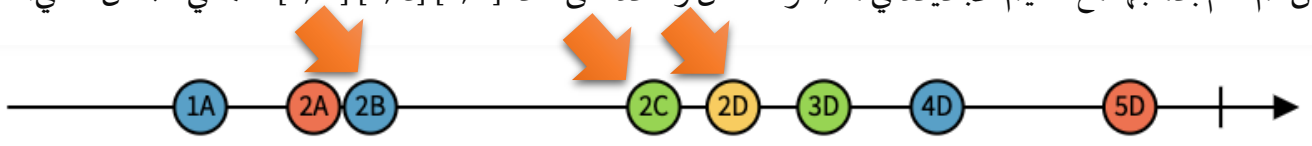
وتم تمرير هذين Observables إلى الدالة combineLatest لكي نقوم بعمل دمج للقيم بحيث يصبح لنا Observable واحد.

ولتسهيل سوف نطلق على Observable الأول o1 والObservable الثاني o2، اما الآن لنشرح بالتفصيل كيف يتم دمج هذه القيم.

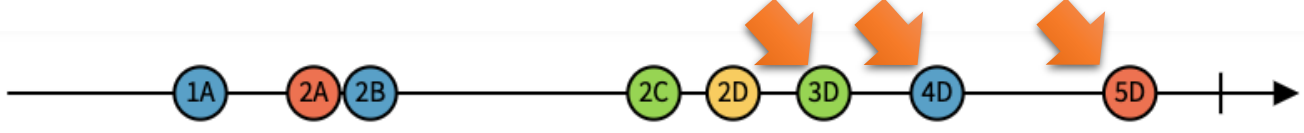
سوف تلاحظ عزيزي المتعلم انه من خلال الشكل السابق قام o1 بعمل emit لأول قيمة وهي 1 ومن ثم الدالة قامت بالانتظار إلى ان تقوم o2 بعمل emit لقيمة ما، وعندما قامت o2 بعمل emit للقيمة A قامت الدالة بدمج هذه القيم بشكل مصفوفة [1, A]، كما في الشكل التالي:

وبعد فترة معينة قامت o1 بعمل emit للقيمة 2 ولكن هنا لم تقوم الدالة combineLatest بانتظار o2 لكي تعمل emit لقيمة مقابلة كما هو الحال في الدالة zip وانما وجدت (آخر) قيمة تم عمل لها emit في o2 ووجدت انها A لذلك قامت بدمج هاتين القيمتين في مصفوفة [2, A]، كما في الشكل التالي:

وبعد فترة زمنية معينة قامت o2 بعمل emit للقيمة B وايضاً بعد فترة زمنية أخرى عملت o2 ايضاً emit لقيم أخرى وهي C وD، كل هذا وo1 لم تعمل emit لأي قيمة لذلك قامت الدالة combineLatest بجلب (آخر) آخر قيمة في o1 وهي 2 ومن ثم قام بدمجها مع القيم الجديدة في مصفوفات كُل واحدة على حدا [2, B] [2, C] [2, D]، كما في الشكل التالي:



وايضاً نفس الأمر تم مع o1 عندما قامت بعمل emit لمجموعة من القيم على فترات زمنية متتابعة وهم 3, 4, 5 حيث قامت الدالة بدمج هذه القيم مع آخر قيمة في o2 وهي D على شكل مجموعة من المصفوفات، كما في الشكل التالي:



ومما سبق يتضح ان هذه الدالة لا تتجاهل أي قيمة من جميع Observables بعكس الدالة zip التي تتجاهل القيم التي ليس لها مقابل في Observables الأخرى.

ولتوضيح لنعطي مثال جيد مصدره على هذا الرابط RxJS Functions (Part 3). combineLatest | by Corey Pyle | Medium

مع بعض التصرف البسيط، وهذا المثال سوف يوضح بإذن الله أغلب هذه المفاهيم الخاصة بهذه الدالة.

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, combineLatest, BehaviorSubject } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private movies$ = new BehaviorSubject([
    { id: 111, title: 'The Land Before Time' },
    { id: 222, title: 'Titan A.E.' },
    { id: 333, title: 'An American Tail' },
  ]);

  private user$ = new BehaviorSubject({ name: 'Corey', favoriteMovieIds: [222] });

  ngOnInit(): void {

    combineLatest([this.movies$, this.user$])
      .pipe(
        map(([movies, user]) => {
          return movies.filter((movie) => {
            return user.favoriteMovieIds.includes(movie.id);
          });
        })
      )
      .subscribe({
        next: (favoriteMovies) => console.log(favoriteMovies),
        error: (error) => console.log(error),
        complete: () => console.log('completed'),
      });

    setTimeout(() => {
      this.user$.value.favoriteMovieIds.push(111);
      this.user$.value.favoriteMovieIds.push(333);
      this.user$.next(this.user$.value);
    }, 3000);

    setTimeout(() => {
      this.user$.value.favoriteMovieIds.splice(
        this.user$.value.favoriteMovieIds.indexOf(222),
        1
      );
```

```
      this.user$.next(this.user$.value);
    }, 6000);

  }
}
```

وتكمن الفكرة الأساسية من هذا المثال، بافتراض انه لدينا كائنين الكائن الأول يحتوي على بيانات مجموعة من الأفلام، والكائن الثاني يحتوي على مجموعة من المستخدمين وفي حالتنا هذه لا يوجد لدينا إلا مستخدم واحد فقط، وهذا المستخدم يحتوي على مصفوفة فرعية للIds الخاصة بكل فيلم، بحيث إذا قام المستخدم بعمل تفضيل للفيلم يتم وضه id الخاص به في هذه المصفوفة وفي حال قام بعمل تفضيل ايضاً بعد فترة لفيلم آخر سوف يتم ايضاً اضافته إلى هذه المصفوفة، وفي حال تم الغاء التفضيل سوف يتم حذفها من المصفوفة.

وهاذين الكائنين قمنا بإنشائهم على شكل Observable باستخدام BehaviorSubject لأننا نُريد ان نعمل emit للقيم من خارج Observable وبنفس الوقت نُريد ان نُعطيها قيم ابتدائية والتي هي في مثالنا هذه بيانات الأفلام وبيانات المستخدمين.

وبما انه الآن أصبح لدينا اثنين Observable واحد باسم movies$ والثاني باسم user$، فنحتاج ان نعمل دمج لهذه Observables في Observable واحد، وهنا تظهر لدينا خيارات متعددة وهي أي دالة نختار لمثل هذه الحالة، هل نختار zip؟ او combineLatest او forkJoin، الخ، لذلك نحتاج ان نفكر قليلا لنعرف احتياجنا، ففي حالتنا هذه سوف يتم عمل emit مباشرة لبيانات مستخدم معين، وايضاً سيتم عمل emit لبيانات مجموعة من الأفلام، وسوف يتم دمجها في Observable واحد، ولكن هنا لو نلاحظ اننا لا نُريد ان نغلق هذا Observable وبنفس الوقت لا يهمنا القيمة المقابلة لكل Observable وانما نُريد ان نراقب أي تغيير يحدث وعند حدوث أي تغيير في أي قيمة من قيم أي Observable فإننا نريد أخذ آخر قيمة من Observable الآخر، وفي حقيقة الأمر هذا ما نريده لأننا نُريد ان نراقب مصفوفة Ids في Observable ذو الاسم user$ وعند حدوث أي تغيير سوف نعمل emit للقيمة الجديدة من خلال Observable ذو الاسم user$ وعند هذه النقطة لا نُريده ان ينتظر قيمة مقابلة جديدة في movies$ وانما آخر قيمة موجودة نريد دمجها مع user$ وهي في حالتنا هذه قائمة الأفلام، لذلك سوف نستخدم الدالة combineLatest، اما الشفرة البرمجية التالية:

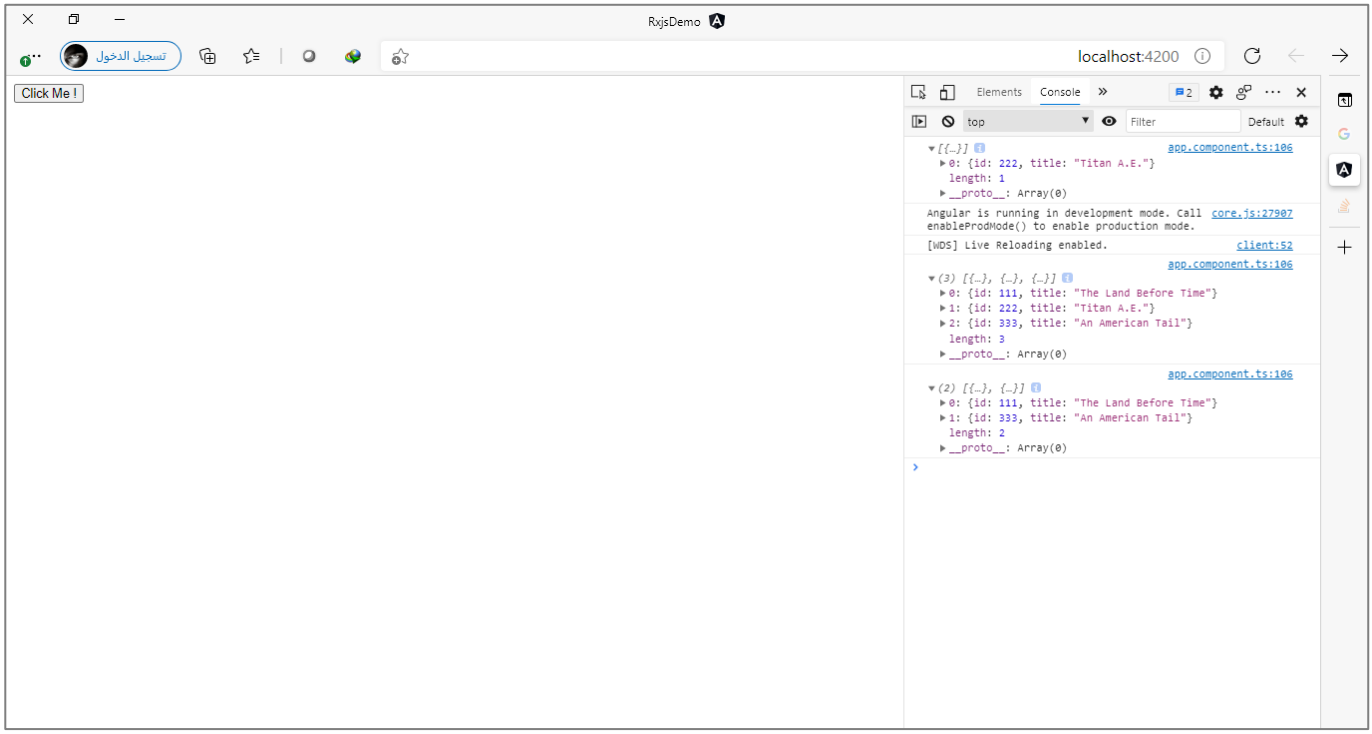
```
combineLatest([this.movies$, this.user$])
  .pipe(
    map(([movies, user]) => {
      return movies.filter((movie) => {
        return user.favoriteMovieIds.includes(movie.id);
      });
    })
  )
  .subscribe({
    next: (favoriteMovies) => console.log(favoriteMovies),
    error: (error) => console.log(error),
    complete: () => console.log('completed'),
  });
```

فلا تهتم لها حالياً وسوف نشرحها بالتفصيل لاحقاً، ووظيفتها بكل بساطة تقوم بعمل ترشيح وفلترة للObservable الأول movies$ بناءً على القيم الجديدة في Observable الثاني user$، حيث تعمل تصفية لقائمة الأفلام بحيث تسترجع فقط الأفلام التي Id الخاص بها مساوٍ لأحد Ids في المصفوفة في user$.

اما الشفرة البرمجية التالية:

```
setTimeout(() => {
  this.user$.value.favoriteMovieIds.push(111);
  this.user$.value.favoriteMovieIds.push(333);
  this.user$.next(this.user$.value);
}, 3000);

setTimeout(() => {
  this.user$.value.favoriteMovieIds.splice(
    this.user$.value.favoriteMovieIds.indexOf(222),
    1
  );
  this.user$.next(this.user$.value);
}, 6000);
```

قمنا بمحاكاة ان المستخدم قام بتفضيل فلمين جديدين التي تحملا Ids => (111, 333) وذلك بعد مرور ثلاث ثواني من تشغيل التطبيق ومن ثم بعد ست ثواني قمنا بمحاكاة ان المستخدم ألغى تفضيل الفيلم ذو id => 222، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ في البداية تم عرض قيمة مبدئية، وبعدها بثلاث ثواني تم اظهار Observable جديد يحمل القيم الجديدة وايضاً بعد مرور ست ثواني ظهر لنا نفس Observable ولكن بقيم جديدة.

وبعد استعراضنا للمثال السابق تبقي علينا ان نعرف كيف نستطيع ان نمرر Observable كأنه كائن إلى هذه الدالة بدلاً من ان نمرره كمصفوفة كما هو موجود في المثال السابق:

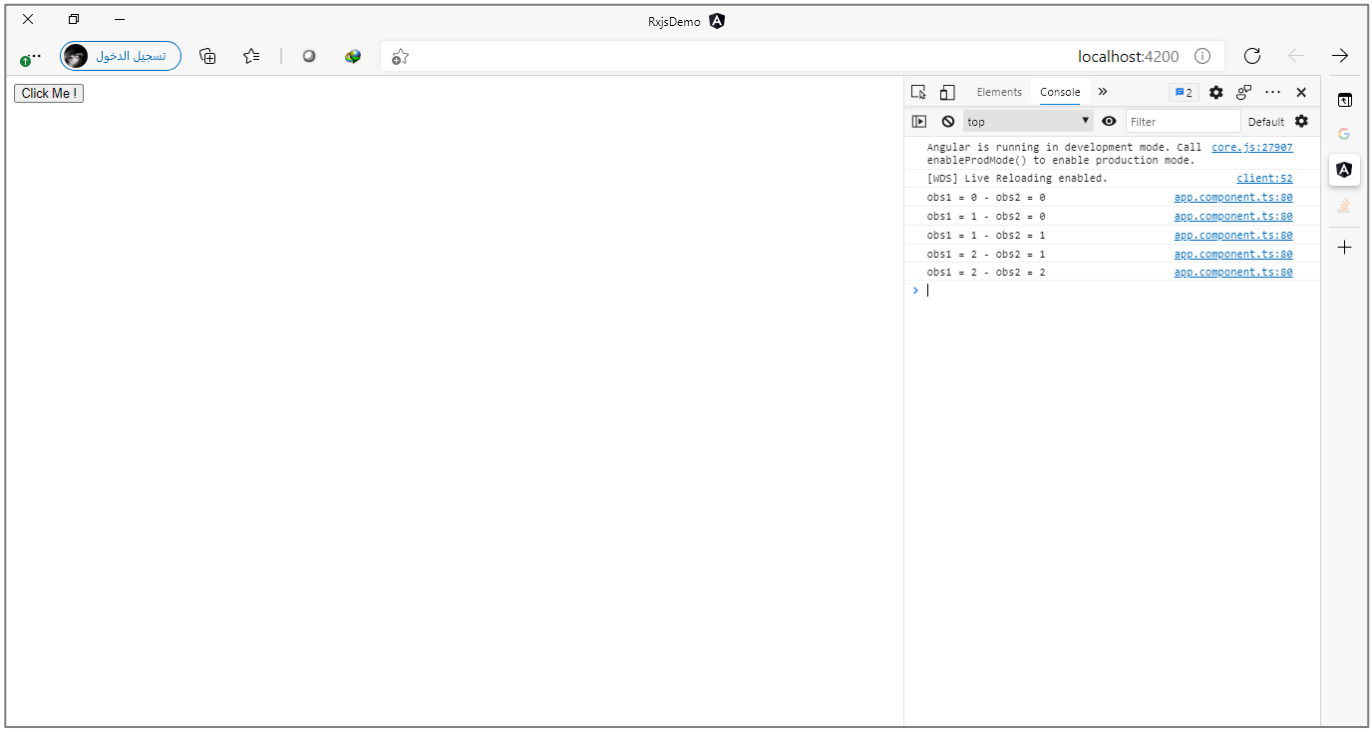
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { combineLatest, timer } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

```
})

export class AppComponent implements OnInit {

  ngOnInit(): void {
    const obs1 = timer(3000, 2000).pipe(take(3));
    const obs2 = timer(5000, 2000).pipe(take(3));
    combineLatest({ o1: obs1, o2: obs2 }).subscribe(({ o1, o2 }) => {
      console.log(`obs1 = ${o1} - obs2 = ${o2}`);
    });
  }

}
```

كما نلاحظ استعضنا عن أسماء Observables => (obs1, obs2) بالـkeys => (o1, o2) بحيث اصبحنا نستعملها بدلاً من أسماء Observables.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



وبذلك قمنا بتغطية اهم المفاهيم في هذه الدالة واما من ناحية النقاط التي تتشابه مع الدالة zip بها مثل طريقة التعامل مع الأخطاء او إذا تم ارسال Observable فارغ وغيره من الأمور الأخرى فلن أعيد شرحها هنا وسأكتفي بما قلناه في الدالة zip وذلك منعاً لتكرار، ولعلها تكون لك كتدريب لك عزيزي المتعلم.

11-2- forkJoin():

وهذه الدالة تجمع من كلا الدالتين السابقتين، والفرق الجوهري المهم أنها لا تقوم بعمل دمج لأي Observable إلا بعدما تكتمل جميع Observables، وعند اكتمال هذه Observables فإنها تأخذ آخر القيم التي تم عمل لها emit، وهي تهتم

بالترتيب بمعنى لو انتهى الObservable الأخير فلا تقوم بعمل emit له قبل الObservables التي قبله وانما تنتظر إلى ان تنتهي جميع الObservables ومن ثم تعمل emit لها بالترتيب بدءً من أول Observable إلى آخر Observable.

وهي مشابهه لدالة combineLatest من حيث يمكن تمرير Observables سواء على شكل مصفوفة او على شكل كائن.

ومن أشهر استخداماتها في حالة أردنا أن نعمل upload او download لمجموعة من الملفات وكل ملف له api خاص به بحيث كل ملف يمثل observable، فعندئذ نستطيع استخدام هذه الدالة لدمج هذه Observables في Observable واحد ومن ثم الانتظار إلى ان تكتمل جميع Observables (أي رفع او تحميل الملفات) ومن ثم تعمل emit لآخر حالة لكل ملف هل نجح ام فشل.

أما الآن لنعطي مثال لتوضيح، وتقوم فكرة المثال انه لدينا اثنين Observables من النوع Subject ومعرف ان أي Observable من النوع Subject لن يعمل له complete بشكل تلقائي مثل دوال of او form او غيرها من هذه الدوال، وبنفس الوقت سوف نضيف أربع أزرار، زرين يقومان بعمل emit لقيم وزرين آخرين يقومان بغلق Observables، كالتالي:

ملف app.component.html

```
<button (click)="incrementObs1()">Increment Observable 1</button>
<button (click)="incrementObs2()">Increment Observable 2</button>
<br />
<br />
{{ obs1Counter }} - {{ obs2Counter }}
<br />
<br />
<button (click)="completeObs1()">Complete Observable 1</button>
<button (click)="completeObs2()">Complete Observable 2</button>
```

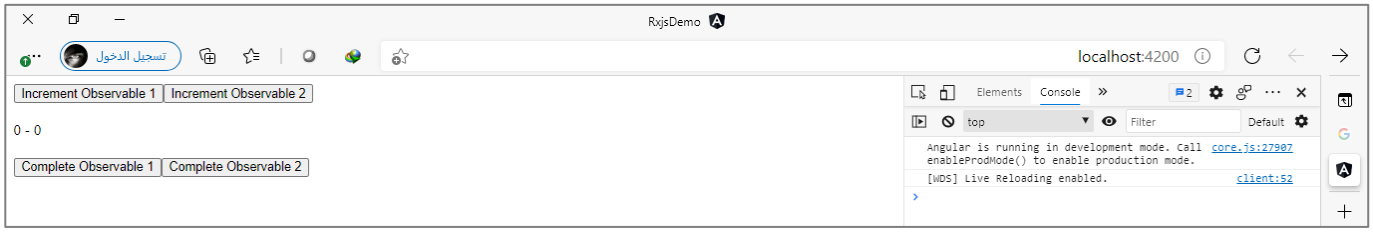
وفي ملف class لهذا component نضيف logic البرمجي التالي:

ملف app.component.ts

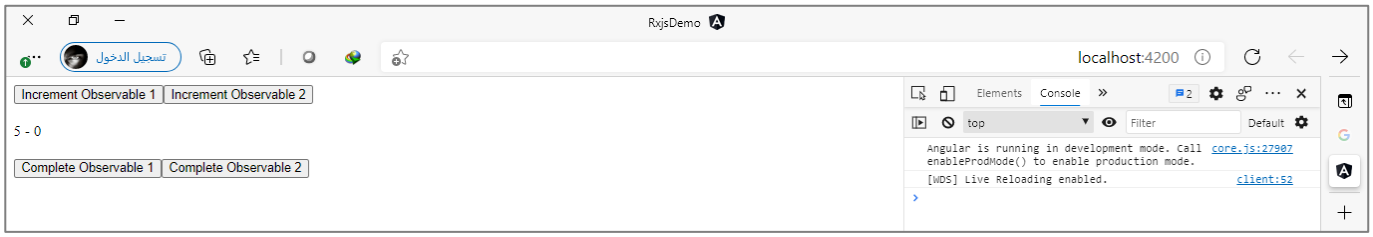
```
import { Component, OnInit } from '@angular/core';
import { forkJoin, Subject } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private obs1 = new Subject<number>();
  private obs2 = new Subject<number>();
  obs1Counter = 0;
  obs2Counter = 0;

  ngOnInit(): void {
    forkJoin([this.obs1, this.obs2]).subscribe(([o1, o2]) =>
```

```
      console.log(`obs1 = ${o1} - obs2 = ${o2}`)
    );
  }

  incrementObs1(): void {
    this.obs1.next(++this.obs1Counter);
  }

  incrementObs2(): void {
    this.obs2.next(++this.obs2Counter);
  }

  completeObs1(): void {
    this.obs1.complete();
  }

  completeObs2(): void {
    this.obs2.complete();
  }
}
```

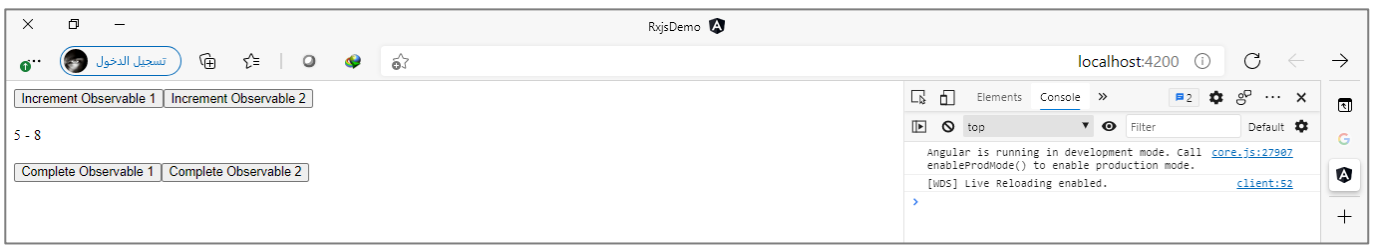
ولنشاهد النتيجة في المتصفح، كالتالي:



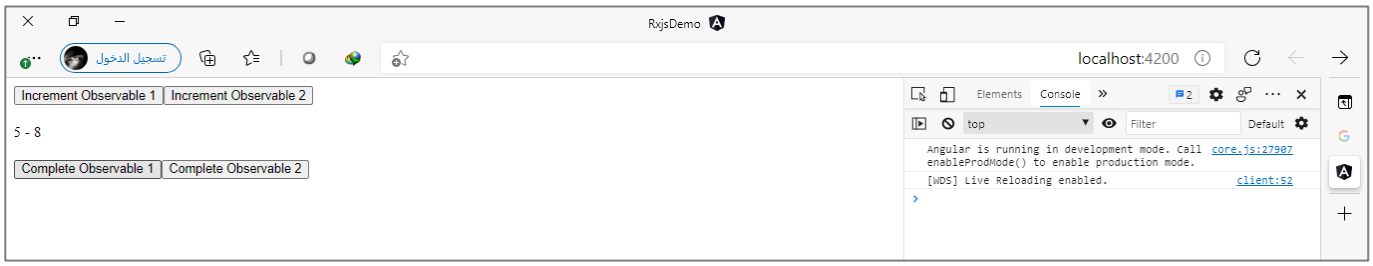
في البداية لنقوم بالضغط على زر Increment Observable 1 عدة ضغطات كالتالي:



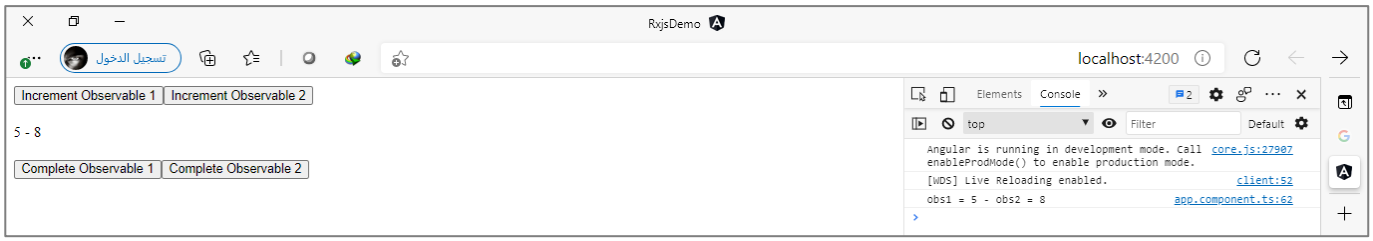
على الرغم من اننا عملنا emit لقيم في obs1 ولكن لم يظهر شيء في console، والسبب ان هذا Observable لم يغلق أي لم يعمل له complete، اما الآن لنقم بالضغط على الزر Increment Observable 2 عدة ضغطات هو الآخر ولنرى النتيجة:



وكما هو واضح نفس النتيجة في الزر الآخر والسبب ان obs2 لم يعمل له complete هو الآخر، أما الآن لنضغط على زر Complete Observable 1، ولنرى النتيجة:



نلاحظ إلى الآن لم يظهر شيء في console والسبب اننا اغلاقنا الObservable الأول ذو الاسم obs1 ولكن الObservable الثاني إلى الآن لم يغلق، لذلك دالة forkJoin لم تعمل، لذلك لنقم بالضغط على الزر الثاني Complete Observable 2 لكي نغلق obs2، ومن ثم لنرى النتيجة كالتالي:



نلاحظ الآن بعدما قمنا بعمل complete لكلا Observables تم تفعيل الدالة forkJoin ومن ثم قامت بقراءة آخر القيم واظهاراها في console.

12-2- concat():

وهذه الدالة ايضاً من دوال دمج مجموعة من Observables في Observable واحد، وما يميزها انها تعمل Subscribe للObservable الأول وعندما يكتمل complete تعمل له unsubscribe ومن ثم تنتقل إلى Observable الثاني وهكذا إلى ان تنتهي من جميع Observables، وتدمجها في Observable واحد.

اما من ناحية استخدامها فيتم استخدامها في حالة أن أردت عزيزي المتعلم أن يتم قراءة جميع Observables بشكل متسلسل، كأن يكون لديك قائمة Playlist وتُريد ان يتم تشغيل القائمة بشكل متسلسل بحيث ينتقل من جزء إلى الجزء الآخر.

ويمكن تقسم تعامل هذه الدالة مع Observables وطرق الدمج، من خلال النقاط التالية:

- تقوم بعمل subscribe لكل observable في كل مرة بشكل متسلسل ومن ثم تقوم بدمج النتائج في Observable واحد، ومن ثم تنتقل إلى Observable الثاني وتقوم بعمل subscribe له بعدما تعمل unsubscribe للObservable الأول، وعند الانتهاء منه تقوم ايضاً بدمج نتائج هذا Observable مع نتائج Observable الأول في Observable واحد، وهكذا تنتقل من Observable إلى الآخر إلى ان تنتهي من جميع Observables، وعندها يتم عمل complete لهذه الدالة.

- في كل مرة يقوم Observable الذي تم عمل subscribe له بعمل emit للقيم الخاصة به، ومن ثم Observable الثاني يعمل emit لقيمه، إلى ان تنتهي من جميع Observables.
- في حالة ان أحد Observables لم ينتهي أي بمعنى لم يتم عمل له complete فإن هذه الدالة لن تعمل هي الأخرى complete ولن تنتقل إلى Observable الذي يلي هذا Observable وإنما سوف تنتظر إلى ان ينتهي، لذلك يجب الانتباه جيداً لمثل هذه الحالات وان استدعت الحالة ان نعمل اغلاق بشكل يدوي لهذا Observable.
- في حالة ان أحد Observables عمل emit لخطأ ما لأي سبب من الأسباب، فإن هذه الدالة لن تعمل subscribed للObservable الذي يلي هذا Observable، بمعنى لو كان لدينا ثلاثة Observables وحدث خطأ في Observable الثاني فإن هذه الدالة سوف تعمل subscribe للObservable الأول وعند الانتهاء منه تنتقل إلى الثاني وتجد انه عمل emit لخطأ ما وعندها تتوقف وتتجاهل باقي Observables الأخرى وتكتفي فقط بالنتائج القادمة من الObservables التي سبقت Observable الذي حدث فيه الخطأ.
- هذه الدالة في المقام الأول لا تهتم أي Observable قام بعمل emit للقيم قبل الآخر، وانما تهتم بترتيب هذه Observable بغض النظر أيهم عمل emit، بمعنى انها تقوم عمل subscribe لجميع Observables بالتسلسل بحسب الترتيب الذي نُعطيه لدالة concat بحيث أول بارامتر يمثل أول Observable وثاني بارامتر يمثل ثاني Observable وهكذا إلى ان يتم الانتهاء من جميع Observables، ولو ان ثاني Observable عمل emit للقيم قبل الأول.
- يتم تمرير الObservables على شكل بارامترات مباشرة إلى هذه الدالة فقط، وليس كمصفوفة أو كائن، كما كُنا نفعل مع الدوال الأخرى، وبنفس الوقت لا يمكن ان نعمل لنتائج Destruction.

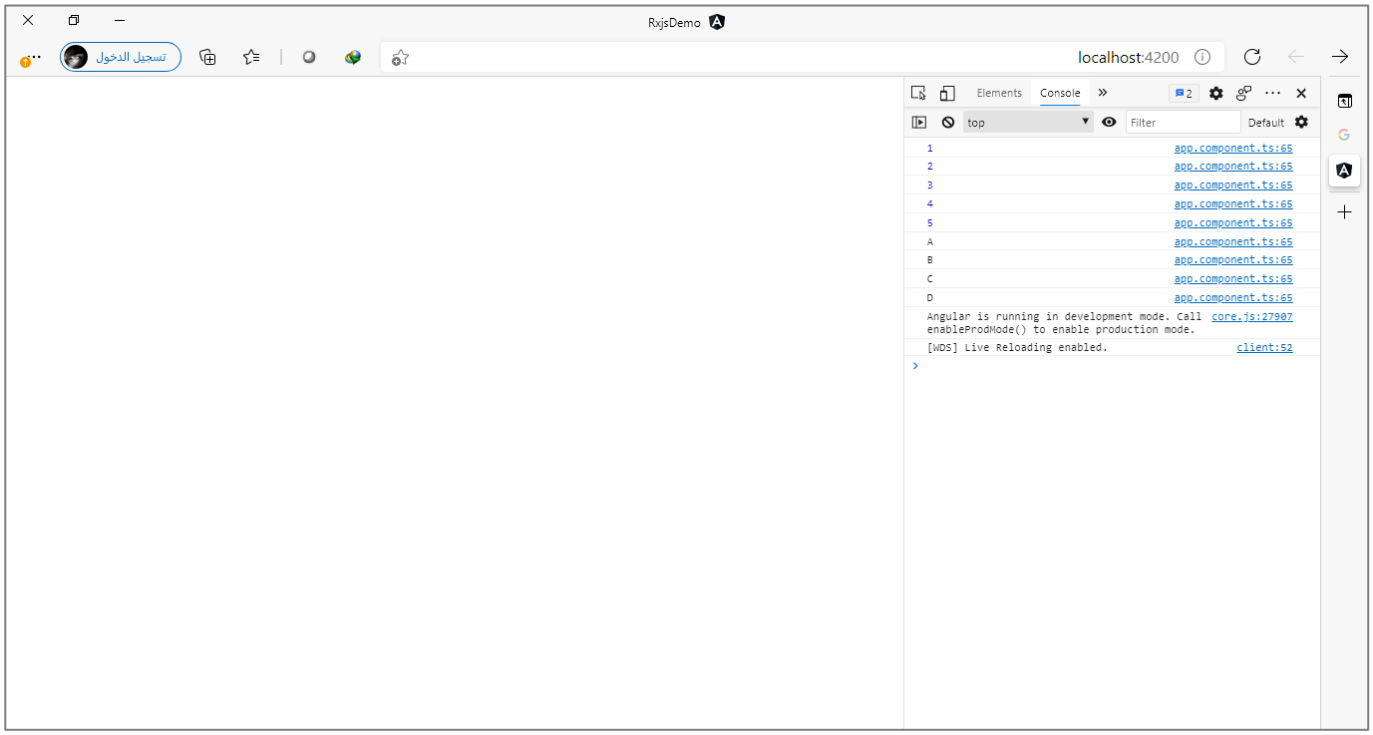
ولتوضيح، لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, concat } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  private obs1 = of(1, 2, 3, 4, 5);
  private obs2 = of('A', 'B', 'C', 'D');

  ngOnInit(): void {
    concat(this.obs1, this.obs2).subscribe((obs) => {
      console.log(obs);
    });
  }
}
```

كما هو واضح من الشفرة البرمجية السابقة، لدينا اثنين Observables واحد اسميته obs1 والثاني باسم obs2، وقمنا بتمريرها إلى الدالة concat على شكل بارامترات بشكل مباشر (ليس مصفوفة او كائن) ومن ثم قما بعمل subscribe وعرضنا الناتج في console.

والمتوقع تكون النتيجة عبارة عن قيم Observable الأول ومن ثم قيم Observable الثاني، كالتالي:



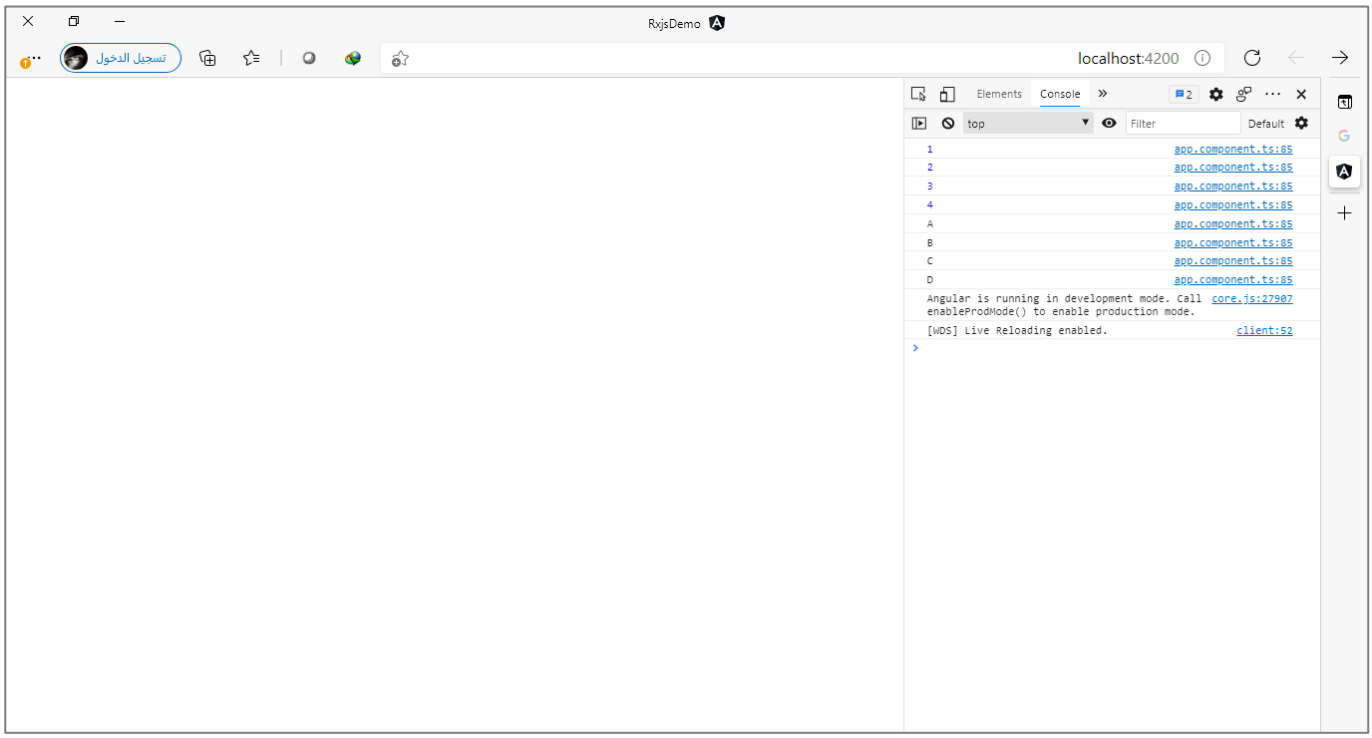
ولنعطي مثال آخر ولنفرض لدينا ثلاثة Observable الأول يقوم بعمل emit لمجموعة من القيم ومن ثم يقوم بعمل complete والثاني يقوم بعمل emit لمجموعة من القيم ولكن **لا يعمل** complete والأخير يقوم بعمل emit ايضاً لقيم أخرى ويقوم بعمل complete، كما في الشفرة البرمجية التالية:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { concat, Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit, AfterViewInit {
  private obs1 = new Observable((observer) => {
    observer.next(1);
    observer.next(2);
    observer.next(3);
    observer.next(4);
    observer.complete();
  });
```

```
  private obs2 = new Observable((observer) => {
    observer.next('A');
    observer.next('B');
    observer.next('C');
    observer.next('D');
  });
  private obs3 = new Observable((observer) => {
    observer.next('Faisal');
    observer.next('Fahd');
    observer.next('Saad');
    observer.complete();
  });

  ngOnInit(): void {
    concat(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete'),
    });
  }

}
```

والمتوقع من الشفرة البرمجية السابقة ان تكون النتيجة، هي قيم Observable الأول لأنه عمل complete ومن ثم قيم Observable الثاني ولكن سوف يتوقف هنا ولن ينتقل إلى Observable الثالث والسبب ان Observable الثاني لم يقم بعمل complete، كما في الشكل التالي:

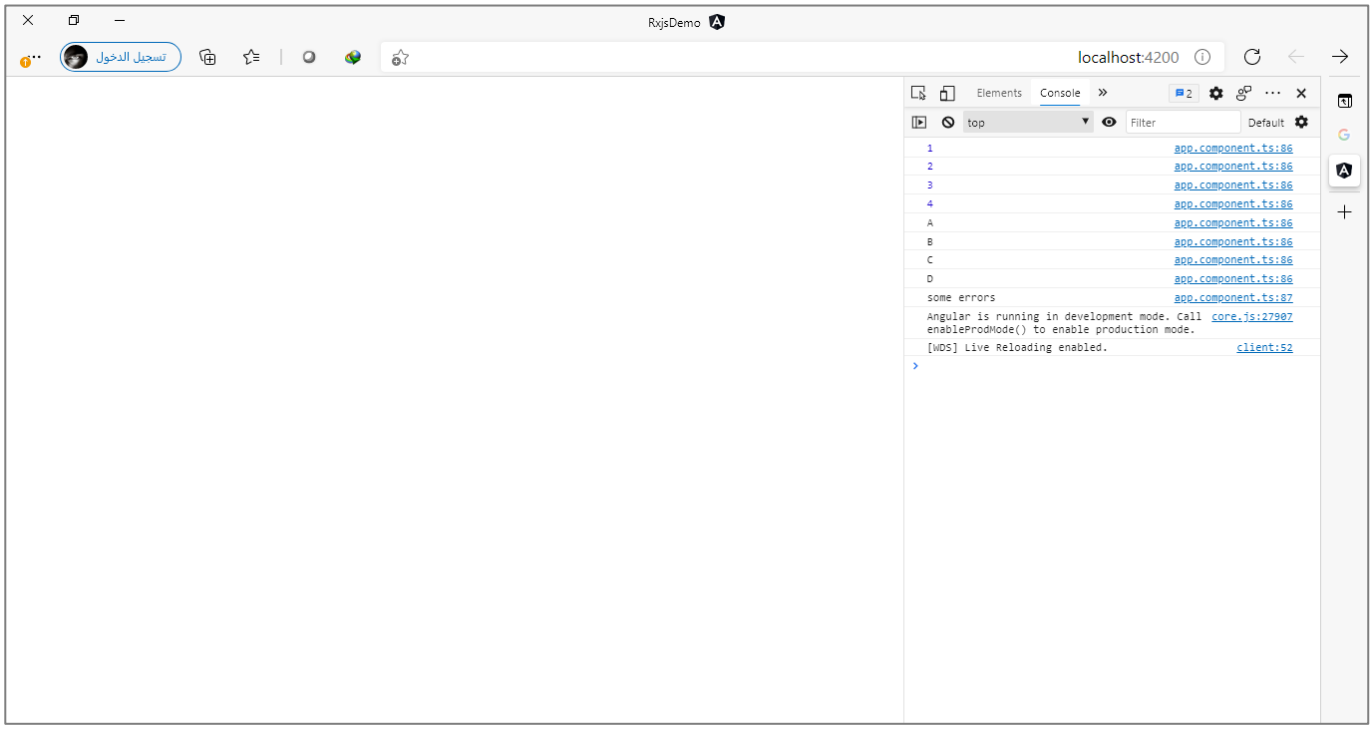


ولنعطي مثال آخر ولكن هنا لنفرض ان Observable الثاني بدلاً من ان لا يعمل complete يقوم بعمل خطأ error، كما في الشفرة البرمجية التالية:

app.component.ts ملف

```
import { Component, OnInit } from '@angular/core';
import { concat, Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit, AfterViewInit {
  private obs1 = new Observable((observer) => {
    observer.next(1);
    observer.next(2);
    observer.next(3);
    observer.next(4);
    observer.complete();
  });
  private obs2 = new Observable((observer) => {
    observer.next('A');
    observer.next('B');
    observer.next('C');
    observer.next('D');
    observer.error('some errors');
  });
  private obs3 = new Observable((observer) => {
    observer.next('Faisal');
    observer.next('Fahd');
    observer.next('Saad');
    observer.complete();
  });

  ngOnInit(): void {
    concat(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete'),
    });
  }
}
```

والنتيجة المتوقعة ان هذه الدالة عندما تجد error سوف تقوم بتجاهل Observables اللاحقة لهذا Observable، ولكن أي Observable سابق سوف تقوم بعرض قيمه بدون أي مشاكل، كما في الشكل التالي:



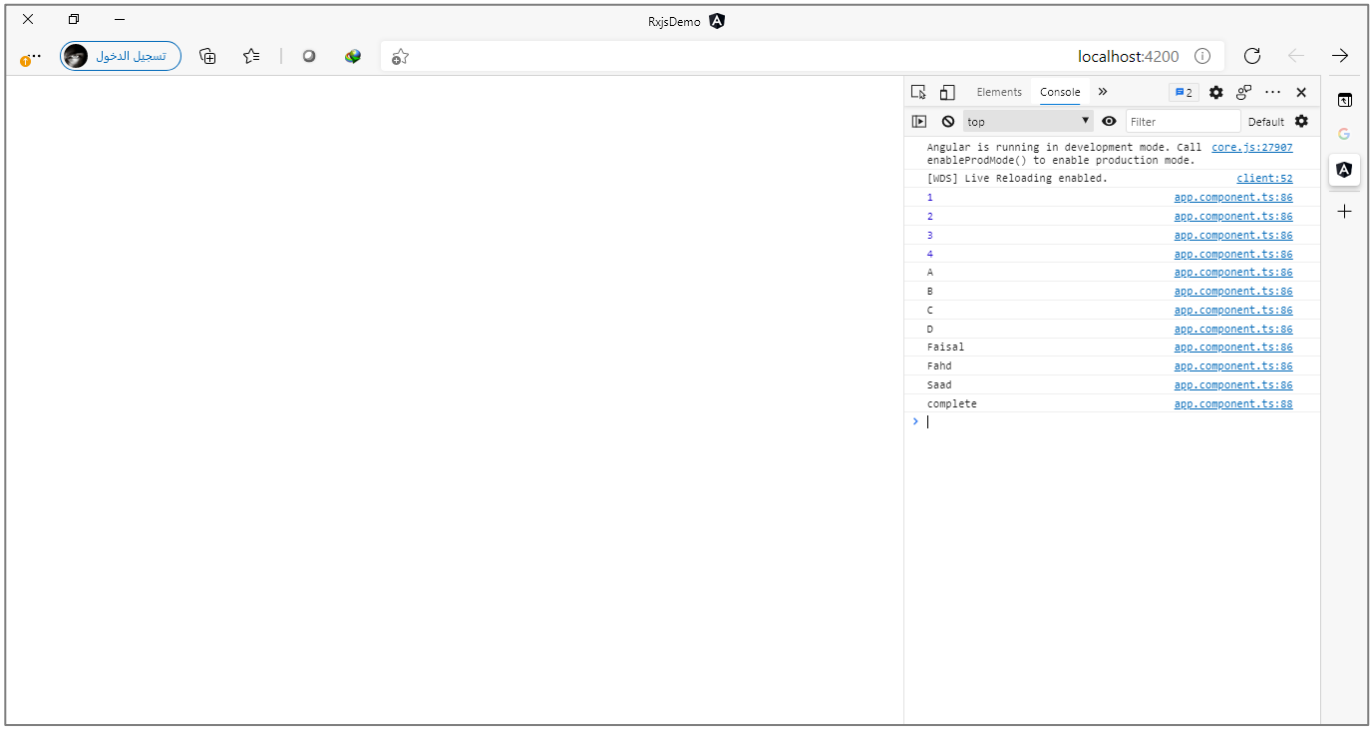
ولنعطي آخر مثال ولنفرض أن جميع Observables الثلاثة قامت بعمل emit وبنفس الوقت complete، ولكن Observable الأول قام بعمل emit للقيم الخاصة به بعد ثلاث ثواني 3000 ميلي ثانية، والObservable الثاني قام بعمل emit بعد خمس ثواني 5000 ميلي ثانية، ولكن الأخير قام بعمل emit للقيم بشكل مباشر وبدون أي تأخير، كما في الشفرة البرمجية التالية:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { concat, Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit, AfterViewInit {
  private obs1 = new Observable((observer) => {
    observer.next(1);
    observer.next(2);
    observer.next(3);
    observer.next(4);
    observer.complete();
  }).pipe(delay(3000));
  private obs2 = new Observable((observer) => {
    observer.next('A');
    observer.next('B');
    observer.next('C');
    observer.next('D');
    observer.complete();
  }).pipe(delay(5000));
```

```
  private obs3 = new Observable((observer) => {
    observer.next('Faisal');
    observer.next('Fahd');
    observer.next('Saad');
    observer.complete();
  });

  ngOnInit(): void {
    concat(this.obs1, this.obs2, this.obs3).subscribe({
      next: (value) => console.log(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete'),
    });
  }
}
```

والمتوقع ان تقوم هذه الدالة بانتظار ثلاث ثواني إلى ان ينتهي Observable الأول من عرض بياناته ومن ثم تنتقل إلى الثاني وتنتظر إلى ان يقوم بعمل emit للقيم الخاصة به، وأخيراً تعرض قيم آخر Observable، لأننا كما أشرت سابقاً هي لا تهتم أي Observable قام بعمل emit قبل الآخر وانما تهتم بترتيب هذه Observable بحسب ما مررناه لدالة، وتكون النتيجة كما في الشكل التالي:



13-2- () merge:

هذه الدالة مشابهه إلى حد كبير مع الدالة concat والفرق الجوهري انها تهتم بالقيم ولا تهتم بترتيب Observables كما في الدالة السابقة، بحيث اول قيمة يتم عمل emit لها هي التي يتم قراءتها في الدالة بغض النظر من أي Observable تأتي هذه القيم، وهكذا بقية القيم.

وفي حال وجد أخطاء في أي Observable فإن هذه الدالة سوف تقوم بقراءة الخطأ، وتكتفي بالقيم السابقة لهذا الخطأ وتتجاهل أي قيمة من أي Observable عمل لها emit، وهي بذلك مشابهه لدالة concat.

ولتوضيح لنعطي المثال التالي: (مصدر المثال على هذا الرابط (123) 24 - timer - range - generate - iif | Rxjs Tutorial In Arabic - YouTube مع بعض التصرف البسيط)

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { merge, of } from 'rxjs';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  public images: string[] = [];
  private img1 = of('https://loremflickr.com/100/100/all').pipe(delay(3000));
  private img2 = of('https://loremflickr.com/100/100/brazil').pipe(delay(1000));
  private img3 = of('https://loremflickr.com/100/100/rio').pipe(delay(2000));
  private img4 = of('https://loremflickr.com/100/100/paris').pipe(delay(5000));
  private img5 = of('https://loremflickr.com/100/100/tree').pipe(delay(900));
  private img6 = of('https://loremflickr.com/100/100/saudi').pipe(delay(4000));
  private img7 = of('https://loremflickr.com/100/100/hero').pipe(delay(500));
  private img8 = of('https://loremflickr.com/100/100/man').pipe(delay(200));
  ngOnInit(): void {
    merge(
      this.img1,
      this.img2,
      this.img3,
      this.img4,
      this.img5,
      this.img6,
      this.img7,
      this.img8
    ).subscribe({
      next: (value) => this.images.push(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete'),
    });
  }
}
```

المثال السابق هو عبارة عن مجموعة من روابط صور عشوائية قمت بتخزينها في متغيرات (img1, ..., img8) بعد تحويلها إلى Observable عن طريق الدالة of، وايضاً قمت بمحاكاة ان بعض الصور قد يأخذ جلبها بعض الوقت لذلك استخدمت الدالة (delay) – سوف نشرحها لاحقاً بإذن الله – ومررت لها الوقت بالملي ثانية.

وأخيراً استخدمت الدالة merge لكي أقوم بدمج هذه Observables المختلفة في Observable واحد ومن ثم كل قيمة من قيم هذا Observable الواحد (روابط الصور) نضيفها إلى مصفوفة اسميتها images لكي نستطيع عرض الصور في ملف template.

اما في ملف template فنضيف markup التالي:

ملف app.component.html

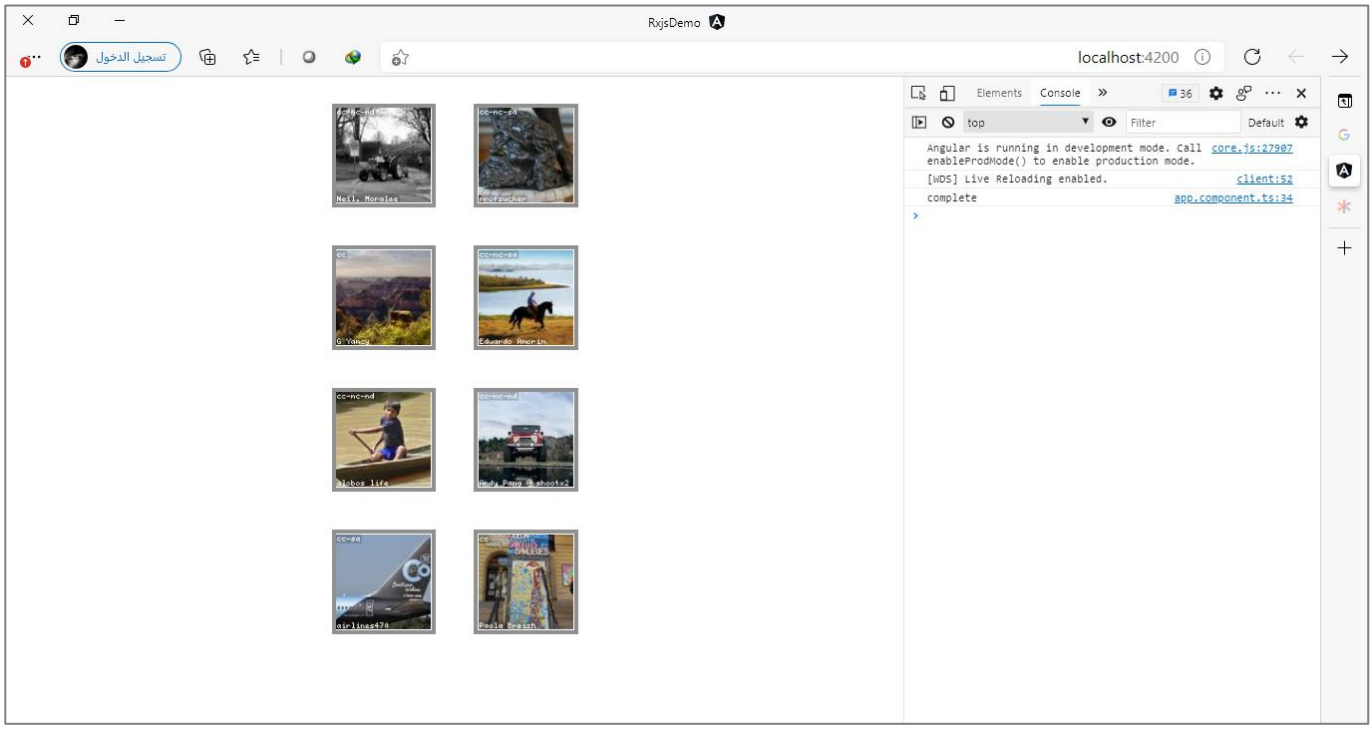
```
<div>
  <img *ngFor="let image of images" [src]="image" />
</div>
```

وأخيراً نُضيف بعض التنسيقات في ملف style، كالتالي:

ملف app.component.scss

```
div {
  box-sizing: border-box;
  width: 300px;
  margin: 0 auto;
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
  align-items: center;
}
img {
  margin: 20px;
  border: 5px solid rgb(145, 145, 145);
}
```

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



سوف تلاحظ عزيزي المتعلم ان او صورة تصل فإن الدالة merge سوف تقوم بعرضها مباشرة في المتصفح، فهي هنا لا تهتم بالترتيب وانما تهتم بأول قيمة تصل سوف يتم عرضها.

وكنوع من التدريب، قم عزيزي المتعلم باستبدال الدالة merge بجميع الدوال الأخرى السابقة مع الانتباه إلى الاختلاف بطريقة وكيفية تمرير القيم او استقبالها في المصفوفة images كما في الدالة zip, combineLatest, forkJoin، وشاهد ما الذي سوف يتغير؟

كما ان هنالك خيار آخر نستطيع تمريره إلى الدالة merge وهو كم Observable نُريد عرضه في نفس الوقت، والقيمة الافتراضية هي 1، أي أول قيمة يتم عمل لها emit هي التي سيتم عرضها، ولكن عندما نقوم بتغيير الرقم إلى اثنين مثلاً ففي هذه الحالة سوف يتم عرض أو قيمتين يتم عمل لها emit وفي حال تأخر القيمة الثانية عن الوصول فسوف تنتظر إلى وصول هذه القيمة ومن ثم تقوم بعرضهما معاً، ومن ثم تنتقل إلى الثالث والرابع وهكذا إلى ان تنتهي من جميع القيم، كما في المثال التالي:

app.component.ts ملف

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { merge, of, Subscription } from 'rxjs';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit, OnDestroy {
  public images: string[] = [];
  public subscription: Subscription;
  private img1 = of('https://loremflickr.com/100/100/all').pipe(delay(3000));
```

```
  private img2 = of('https://loremflickr.com/100/100/brazil').pipe(delay(1000));
  private img3 = of('https://loremflickr.com/100/100/rio').pipe(delay(2000));
  private img4 = of('https://loremflickr.com/100/100/paris').pipe(delay(5000));
  private img5 = of('https://loremflickr.com/100/100/tree').pipe(delay(900));
  private img6 = of('https://loremflickr.com/100/100/saudi').pipe(delay(4000));
  private img7 = of('https://loremflickr.com/100/100/hero').pipe(delay(500));
  private img8 = of('https://loremflickr.com/100/100/man').pipe(delay(200));
  ngOnInit(): void {
    this.subscription = merge(
      this.img1,
      this.img2,
      this.img3,
      this.img4,
      this.img5,
      this.img6,
      this.img7,
      this.img8,
      2
    ).subscribe({
      next: (value) => this.images.push(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete'),
    });
  }
  ngOnDestroy(): void {
    this.subscription.unsubscribe();
  }
}
```

14-2- race() :

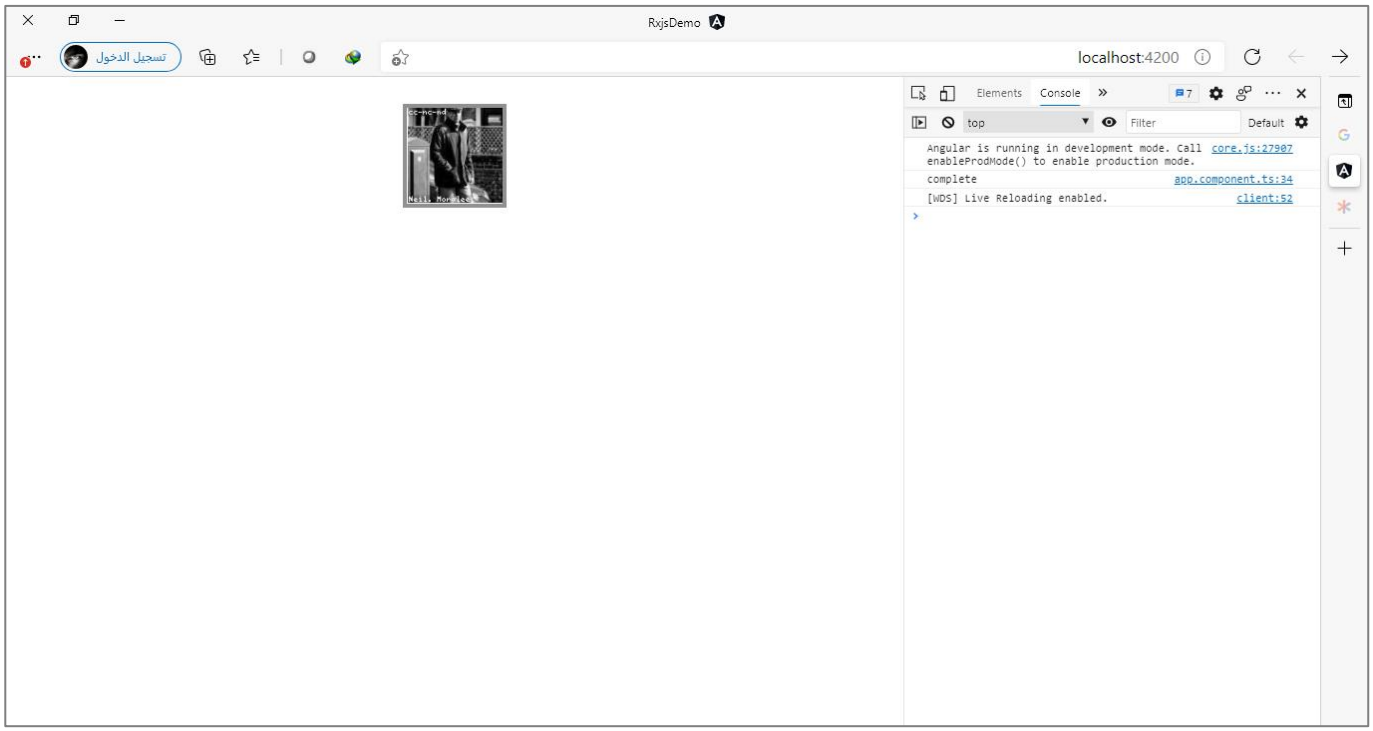
وهذه الدالة لها من أسمها نصيب فهي تُعامل الObservables كأنهم في حلبة سباق بحيث أول Observable ينتهي هو الذي سوف تقوم بقراءته والتعامل معه وتتجاهل باقي Observables الأخرى، كما في المثال التالي:

ملف app.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { of, Subscription, race } from 'rxjs';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit, OnDestroy {
  public images: string[] = [];
  public subscription: Subscription;
  private img1 = of('https://loremflickr.com/100/100/all').pipe(delay(3000));
  private img2 = of('https://loremflickr.com/100/100/brazil').pipe(delay(1000));
  private img3 = of('https://loremflickr.com/100/100/rio').pipe(delay(2000));
  private img4 = of('https://loremflickr.com/100/100/paris').pipe(delay(5000));
```

```
  private img5 = of('https://loremflickr.com/100/100/tree').pipe(delay(900));
  private img6 = of('https://loremflickr.com/100/100/saudi').pipe(delay(4000));
  private img7 = of('https://loremflickr.com/100/100/hero').pipe(delay(500));
  private img8 = of('https://loremflickr.com/100/100/man').pipe(delay(200));
  ngOnInit(): void {
    this.subscription = race(
      this.img1,
      this.img2,
      this.img3,
      this.img4,
      this.img5,
      this.img6,
      this.img7,
      this.img8
    ).subscribe({
      next: (value) => this.images.push(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete'),
    });
  }

  ngOnDestroy(): void {
    this.subscription.unsubscribe();
  }
}
```

اما النتيجة في المتصفح، فتكون كالتالي:



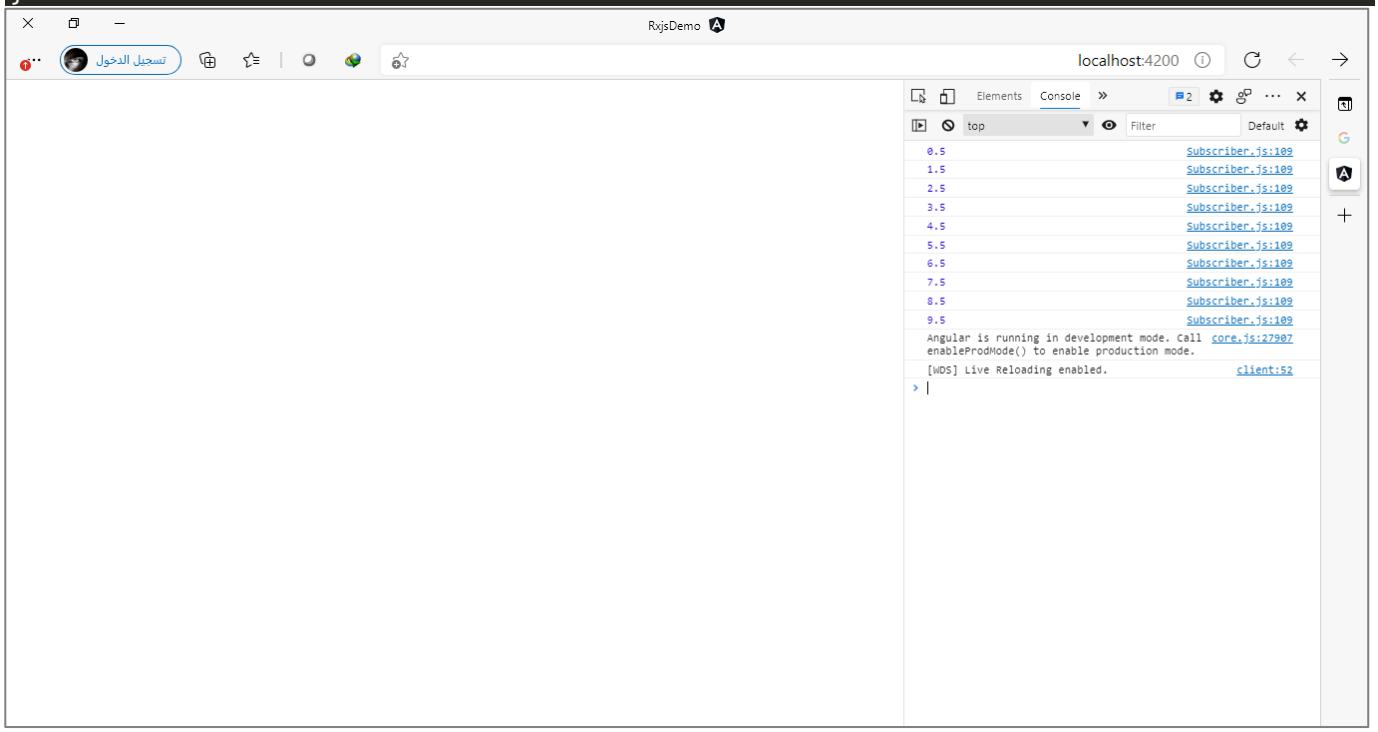
نلاحظ تم عرض صورة واحدة فقط وهي اول صورة تم عمل لها emit قبل Observables الأخرى.

15-2- ()rang:

هذه الدالة من دوال العمليات الحسابية وتقوم بإنشاء Observable من مجموعة من القيم الرقمية، وهذه القيم هي عبارة عن مدى لقيمة ابتدائية وقيمة نهائية ومهمة هذه الدالة إيجاد القيم بين هذين القيمتين، وفي كل مرة يتم الزيادة بواحد، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { range } from 'rxjs';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit, OnDestroy {
  ngOnInit(): void {
    range(0.5, 10).subscribe(console.log);
  }
}
```



نلاحظ مررنا للدالة قيمة مبدئية وهي 0.5 وقيمة نهائية وهي 10 وقامت الدالة بإيجاد المدى range بين هذين القيمتين بزيادة واحد في كل مرة، أي 0.5, 1.5, 2.5, ..., 9.5، اما من ناحية استخدامات هذه الدالة فنستطيع استخدامها على سبيل المثال لعرض pagination او ارقام الصفحات. كما في الصورة التالية:

16-2- () iif:

وهي من الدوال الشرطية، وتستقبل أثنين Observable وبناءً على شرط معين وفي حال تحققه تقوم هذه الدالة بعمل subscribe لأحد هذين Observables.

وتستقبل هذه الدالة ثلاث بارامترات الأول هو الشرط ويكون على شكل دالة بحيث تُعيد قيمة منطقية true او false والثاني هو Observable الأول ويتم عمل subscribe له في حال تحقق الشرط وأرجع القيمة true والبارامتر الأخير هو Observable الثاني ويتم عمل subscribe له في حال عدم تحقق الشرط أي أرجع القيمة false.

ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { iif, of } from 'rxjs';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  ngOnInit(): void {
    iif(() => 10 % 2 === 0, of('عدد زوجي'), of('عدد فردي')).subscribe(
      console.log
    );
  }
}
```

كما نلاحظ قمنا بتمرير ثلاث بارامترات الأول هو الشرط وهو عبارة عن دالة method تُعيد قيمة منطقية، وهذه الدالة تستطيع كتابة محتوياتها منفصلة ومن ثم تمريرها هنا كبارامتر او كما فعلنا في هذا المثال قمنا بكتابة محتوى الدالة مباشرة. وفي حال تحقق هذا الشرط يتم قراءة وعمل subscribe للObservable الأول وفي حال عدم تحقق الشرط سيتم قراءة الObservable الثاني، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:

وفي حال قمنا بتغيير القيمة من 10 إلى مثلاً 11، لكيلا يتحقق الشرط، فإن النتيجة تكون كالتالي:

وفي حال عدم تمرير Observable الثاني فإن هذه الدالة سوف تُعيد EMPTY إذا لم يتحقق الشرط.

2-17- partition() :

وهي من الدوال المهمة ولها استخداماتها، ووظيفتها هي تجزئة أي Observable إلى جزئيين بناءً على شرط معين، مع إمكانية التعامل مع كلا الجزئيين.

وتستقبل بارامترين الأول هو Observable الذي نُريد تجزئته والثاني عبارة عن دالة Method تُعيد قيمة منطقية وتحتوي على الشرط الذي بموجبه تقوم هذه الدالة بتقسيم هذا Observable، وهي ايضاً Generic ولكن ليس اجباري.

ولتوضيح لنفرض انه لدينا مصفوفة من الكائنات على شكل Observable بحيث يحتوي كل كائن على بيانات مجموعة من المطورين، ونُريد ان نقسم هذا Observable إلى قسمين او جزئيين او بمسمة آخر إلى أثنين Observables بحيث الجزء الأول يحتوي على بيانات مطوري Angular والجزء الثاني يحتوي على بيانات المطورين الآخرين.

app.component.ts ملف

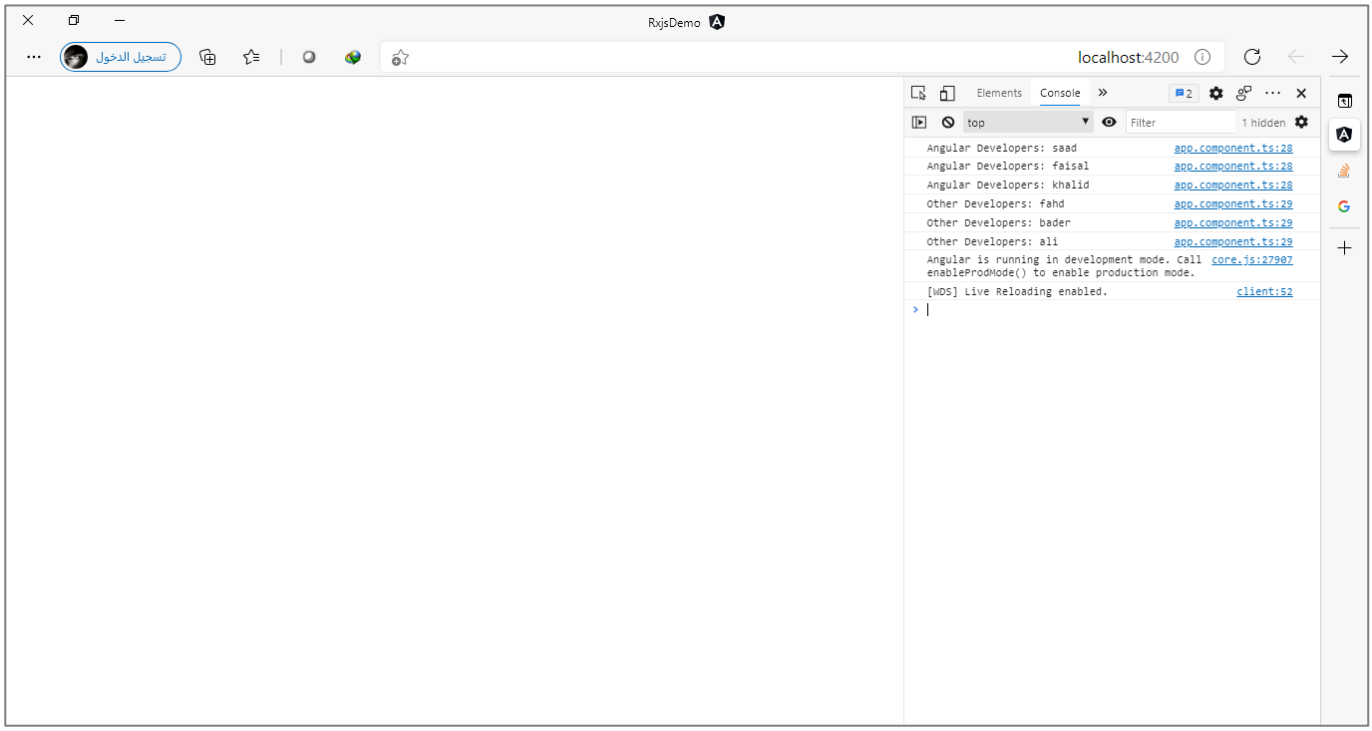
```
import { Component } from '@angular/core';
import { from, partition } from 'rxjs';

export interface Developer {
  name: string;
  knowledge: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private obs: Observable<Developer> = from([
    { name: 'saad', knowledge: 'angular' },
    { name: 'fahd', knowledge: 'react' },
    { name: 'bader', knowledge: 'vue' },
    { name: 'faisal', knowledge: 'angular' },
    { name: 'khalid', knowledge: 'angular' },
    { name: 'ali', knowledge: 'c#' },
  ]);

  ngOnInit(): void {
    const [angular, other] = partition<Developer>(
      this.obs,
      (e) => e.knowledge === 'angular'
    );
    angular.subscribe(({ name }) => console.log(`Angular Developers: ${name}`));
    other.subscribe(({ name }) => console.log(`Other Developers: ${name}`));
  }
}
```

في البداية قمت بتعريف interface لوصف البيانات – يُفترض بك عزيزي المتعلم ان يكون لديك معرفة بها فهي من تقنيات Typescript ولكن سوف اشرحها على عُجالة بإذن الله في الفصل الخاص بالـHttpClient – وايضاً قمت بتحويل البيانات إلى Observable باستخدام الدالة from وتخزين القيم في المتغير obs.

وبعدها استخدمنا الدالة partition وبما انها Generic لكن بشكل اختياري وليس الزامي، فعندها تستطيع ان تحدد نوعها من عدمه ولكن هنا أحببت ان احدد النوع وهو نفس نوع Observable الداخل لهذه الدالة والمتمثل بالمتغير obs، ومن ثم مررنا البارامترات لهذه الدالة وهم obs ودالة تمثل الشرط الذي بناءً عليه تقوم هذه الدالة بتجزئة هذا Observable.

بما ان هذه الدالة سوف تُعيد اثنين Observables لذلك قمنا بعمل destruction لها في متغيرين اسميتهما angular, other، بحيث كل واحد منهما اصبح يمثل Observable منفصل عن الآخر، مع العلم ان المتغير الأول angular سوف يضم أي قيمه حققت الشرط والمتغير الآخر سوف يحمل باقي القيم.

وأخيراً قمنا بعمل subscribe وعرضنا القيم في شاشة console، بحيث تكون النتيجة، كالتالي:



18-2- generate():

وتقوم فكرة هذه الدالة بشكل مبسط مقام حلقة التكرار For التي لا تخلو أي لغة برمجة منها، ولا كنها هنا نقوم بإعادة القيم من التكرار على شكل Observable.

وتستقبل هذه الدالة كائن Object يحتوي على أربع أجزاء رئيسية الجزء الأول وهو القيم الابتدائية initialState على شكل خاصية Property، والجزء الثاني هو عبارة عن الشرط الذي يقف عنده التكرار condition على شكل دالة، والجزء

الثالث هو مقدار الزيادة في التكرار iterate على شكل دالة هو الآخر، اما الأخير فهو دالة ايضاً ونستفاد منه لتحكم بنتيجة كل قيمة خارجة من هذا Loop (أن صحة التسمية) بحيث نقوم بالتعديل عليها بحسب الحاجة.

ولتوضيح لنعطي المثال التالي: مصدر المثال (RxJS - generate)

app.component.ts ملف

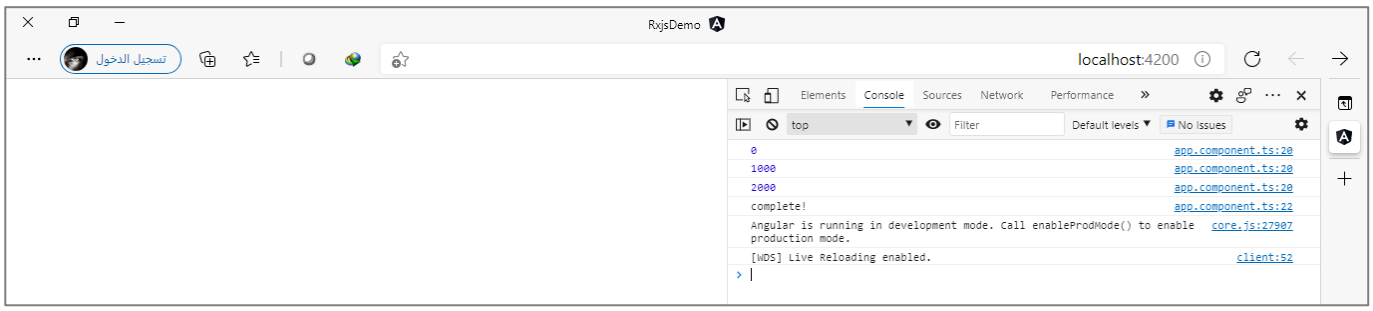
```
import { Component } from '@angular/core';
import { generate } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  ngOnInit(): void {
    const result = generate({
      initialState: 0,
      condition: (value) => value < 3,
      iterate: (value) => value++,
      resultSelector: (value: number) => value * 1000,
    });

    result.subscribe({
      next: (value) => console.log(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete!'),
    });
  }
}
```

نلاحظ مررنا كائن إلى الدالة generate وأول جزء وهو initialState القيمة الابتدائية وضعناها بصفر، والجزء الثاني condition وهو عبارة عن دالة تستقبل قيمة على شكل بارامتر اسميته value، وكان الشرط استمر بعمل Loop إلى ان تصل قيمة value أقل من 3 أي تساوي 2، والجزء الثالث عبارة مقدار الزيادة في value وجعلناها تزدادا بمقدار واحد كل مرة أي في البداية، والجزء الرابع والأخير هو التلاعب والتحكم بالنتيجة الخارجة وكل الذي عملناه قمنا بضرب الناتج في 1000.

لذلك عزيزي المتعلم سوف تلاحظ ان كل قيمة سوف تمر بثلاث مراحل لو استثنينا القيمة الابتدائية، ولو حاولنا متابعة سير هذه الدالة سنجد انها في البداية تأخذ القيمة الابتدائية وهي في حالتنا هذه صفر ومن ثم تتأكد من شرط إيقاف هذا Loop وستجد ان الصفر اصغر من 3 ومن ثم تمرر هذه النتيجة إلى الجزء الأخير وستجد السطر البرمجي وهو ضرب النتيجة في 1000 وبطبيعة الحالة النتيجة ستكون صفر، ومن ثم تُزيد قيمة value بواحد، وتُعيد نفس الخطوات وستكون النتيجة 1000 (لأن 1000 ضرب واحد يساوي 1000 ☺) ومن ثم تُزيد قيمة value بواحد فتصبح قيمته 2 وتعيد نفس الخطوات وستكون النتيجة 2000، ومن ثم تُزيد value بواحد فتصبح قيمته 3 ولكن هنا عن وصولها إلى

جزء الشرط ستجد ان الشرط تحقق فقيم value ليست اصغر من 3 وانما تساويها، لذلك تخرج من Loop، وتُرسل النتائج على شكل Observable وتعمل complete لها.

ولتأكيد لنشاهد النتيجة في المتصفح، كالتالي:



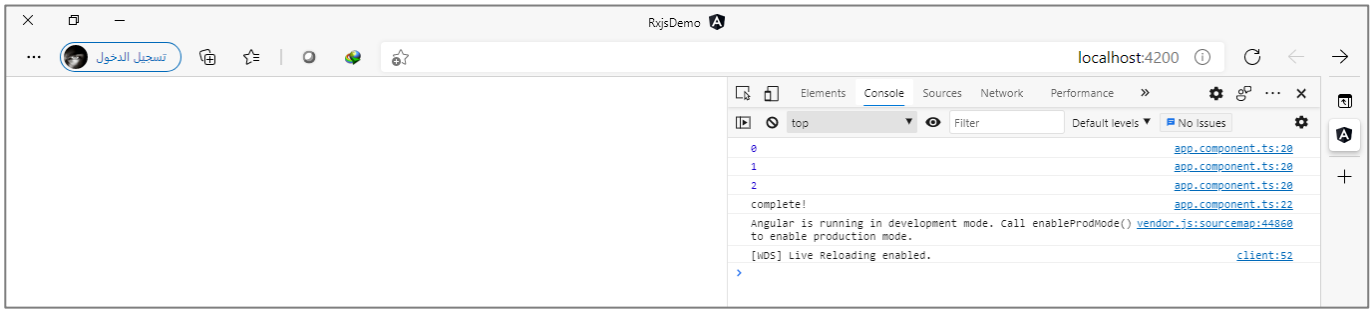
نلاحظ النتيجة كما هو متوقع.

وايضاً نستطيع الاستغناء عن resultSelector، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { generate } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  ngOnInit(): void {
    const result = generate({
      initialState: 0,
      condition: (value) => value < 3,
      iterate: (value) => value++,
    });

    result.subscribe({
      next: (value) => console.log(value),
      error: (error) => console.log(error),
      complete: () => console.log('complete!'),
    });
  }
}
```

وتكون النتيجة، كالتالي:



19-2- throwError():

وهي من دوال معالجة الأخطاء التي تقدمها لنا مكتبة Rxjs، حيث تتيح للمطورين في حالة وجود أي خطأ في stream البيانات (Observable) ان نعمل معالجة لهذا الخطأ على شكل Observable، بمعنى انها تقوم بإنشاء Observable جديد وعن طريقه نستطيع ان نمرر نوع الخطأ ورسالة الخطأ و ...الخ، ومن ثم نستقبل هذه الرسالة في Subscriber ونعرضها للمستخدم.

وسابقاً كانت هذه الدالة تستقبل رسالة الخطأ فقط ولكن هذا الشكل تم عمل له deprecate وأصبحت تستقبل دالة على شكل بارامتر (تُسمى Errors Factory) وعن طريق هذه الدالة نستطيع معالجة الأخطاء كيفما نُريد.

كما انها تقوم بإرسال Observable جديد لكل Subscription جديد، ولو كانت نفس القيم (نفس رسالة الخطأ او رقم الخطأ او ...الخ)، أي انها Unicast Observable.

ولتوضيح لنعطي المثال التالي: المصدر (RxJS - throwError) مع بعض التعديل البسيط.

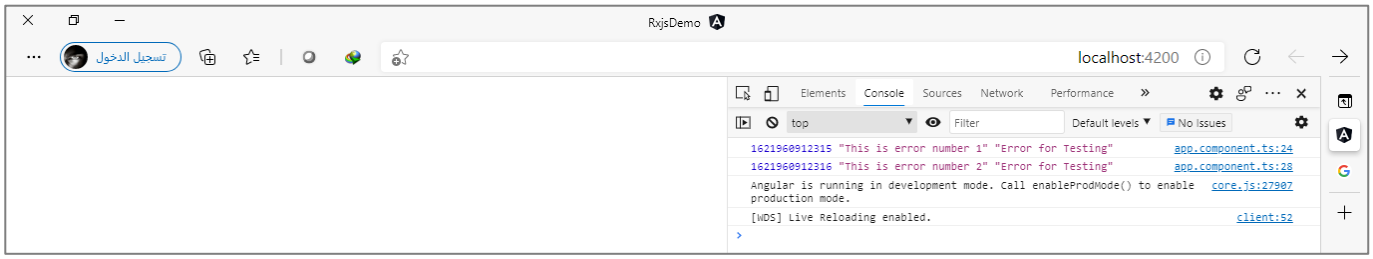
ملف app.component.ts

```
import { Component } from '@angular/core';
import { throwError } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {

  ngOnInit(): void {
    let errorCount = 0;

    const errorWithTimestamp$ = throwError(() => {
      const error: any = new Error(`This is error number ${++errorCount}`);
      error.info = 'Error for Testing';
      error.timestamp = Date.now();
      return error;
    });

    const subscription1 = errorWithTimestamp$;
```

```
    subscription1.subscribe({
      error: (err) => console.log(err.timestamp, err.message, err.info),
    });

    const subscription2 = errorWithTimestamp$;
    subscription2.subscribe({
      error: (err) => console.log(err.timestamp, err.message, err.info),
    });
  }

}
```

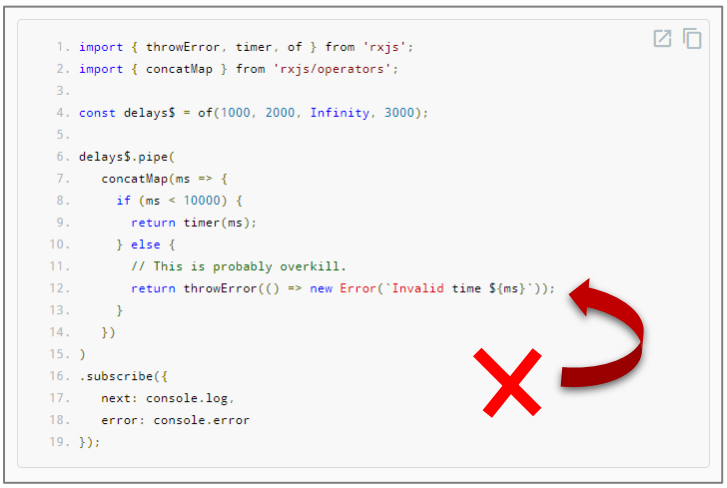
نلاحظ عزيزي المتعلم هذه الدالة ومررنا لها دالة أخرى كبارامتر وبداخلها حددنا شكل ونوع وبعض البيانات التي نُريدها عن هذا الخطأ، ومن ثم في كل subscription جديد يحدث فإن هذه الدالة تقوم بعمل emit لـObservable جديد.

اما الآن لنشاهد النتيجة في المتصفح:

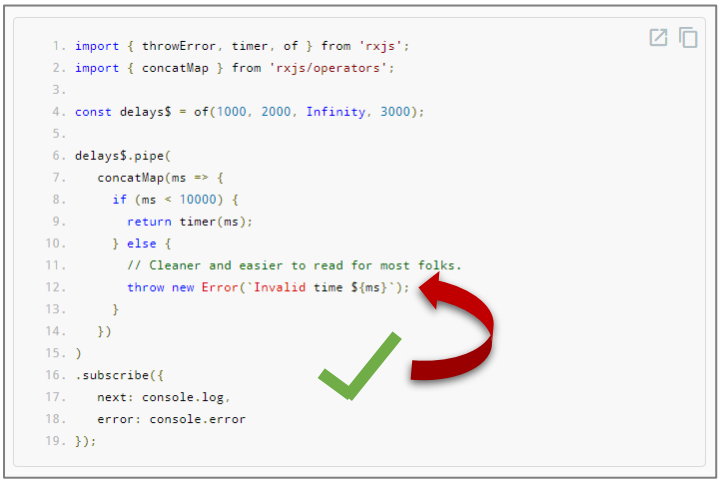


ويجب الاشارة لنقطة مهمة وهي ان هذه الدالة في العادة لا يتم استخدامها بهذه الطريقة وانما تُستخدم مع دوال pipe والتي سوف نتكلم عنها في القسم الثاني من هذه الدوال، بحيث نكتشف الخطأ في Stream البيانات عن طريق دوال معينة سنتطرق لها لاحقاً ومن ثم نستخدم هذه الدالة لمعالجة هذا الخطأ.

لذلك ينصح الموقع الرسمي لمكتبة Rxjs بانه في حالة كان لدينا معالجة بسيطة في الخطأ، كأن نريد ان نُرسل رسالة فقط، فعندها نستخدم الطريقة التقليدية throw new Error() بدلاً من المبالغة بإنشاء Observable جديد باستخدام الدالة throwError لكي نقوم بمعالجة بسيطة للخطأ، ولتوضيح لنعطي المثال التالي:



```
import { throwError, timer, of } from 'rxjs';
import { concatMap } from 'rxjs/operators';

const delays$ = of(1000, 2000, Infinity, 3000);

delays$.pipe(
  concatMap(ms => {
    if (ms < 10000) {
      return timer(ms);
    } else {
      // This is probably overkill.
      return throwError(() => new Error(`Invalid time ${ms}`));
    }
  })
)
.subscribe({
  next: console.log,
  error: console.error
});
```



20-2- onErrorResumeNext():

وهذه ايضاً من دوال معالجة الأخطاء ولكن تفرق عن الدالة throwError بانها لا توقف Stream البيانات او ما يسمى Observable وانما تنتقل إلى الObservable التالي في حال وجوده.

ومن هذا المنطلق نقول ان هذه الدالة تستقبل عدد من Observables على شكل بارامترات ومن ثم في حال وجود خطأ او اكتمال أحد Observables فإنها تنتقل الى الObservable الذي يليه وهكذا إلى ان تنتهي من جميع Observables المُمررة لها، وفي حال وجود أي Observable لم يحدث فيه أي خطأ وايضاً لم يتم يكتم أي لم يعمل له complete فإن هذه الدالة لن تنتقل للObservable الذي يليه، لأن من شروط انتقاله هو اما خطأ او اكتمل.

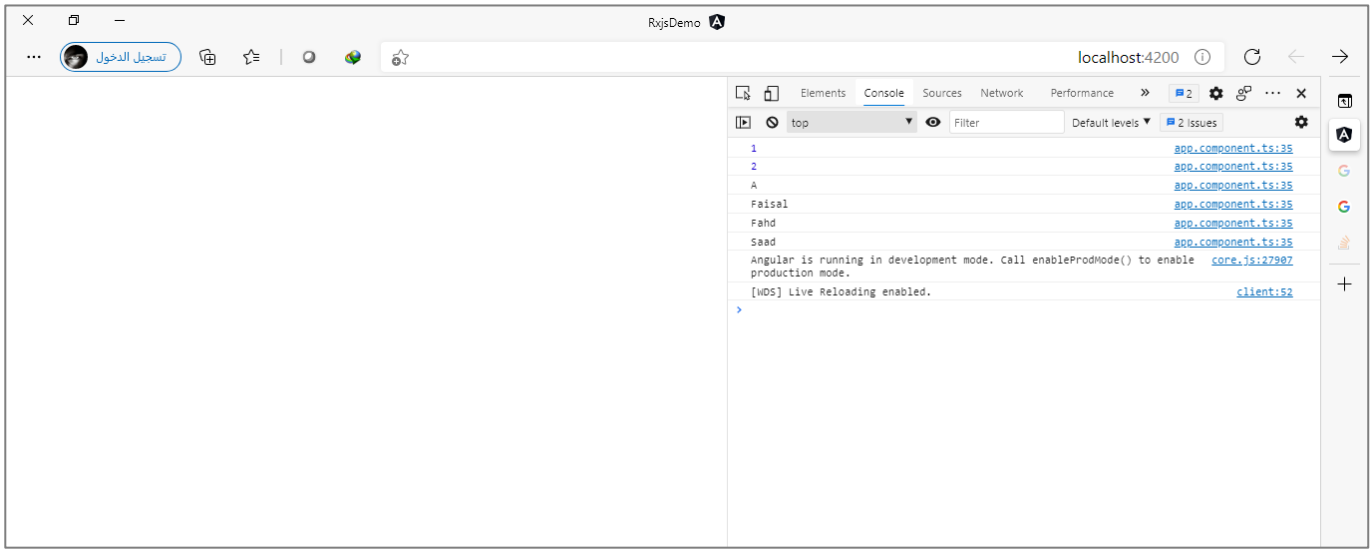
ولتوضيح لنعطي المثال التالي:

app.component.ts ملف

```
import { Component } from '@angular/core';
import { Observable, onErrorResumeNext } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  // tslint:disable-next-line: use-lifecycle-interface
  ngOnInit(): void {
    const obs1 = new Observable((observer) => {
      observer.next(1);
      observer.next(2);
      observer.complete();
      observer.next(3);
    });
    const obs2 = new Observable((observer) => {
      observer.next('A');
      observer.error('error');
      observer.next('B');
      observer.next('C');
```

```
    });
    const obs3 = new Observable((observer) => {
      observer.next('Faisal');
      observer.next('Fahd');
      observer.next('Saad');
    });
    const obs4 = new Observable((observer) => {
      observer.next(111);
      observer.next(222);
      observer.next(333);
      observer.error('Errors');
    });
    onErrorResumeNext(obs1, obs2, obs3, obs4).subscribe((value) => {
      console.log(value);
    });
  }
}
```

نلاحظ ان obs1 عمل emit لقيمتين ثم أكتمل، اما الثاني obs2 عمل emit لقيمة واحدة ثم عمل error اما الثالث obs3 عمل emit لجميع القيم ولكن لم يكتمل او يعمل error والرابع obs4 عمل emit للقيم الخاصة به ثم عمل error، وأخيراً قمنا بتمرير جميع هذه Observables إلى الدالة onErrorResumeNext، تما الآن لنشاهد النتيجة في المتصفح، كالتالي:



كما تلاحظ عزيزي المتعلم قامت هذه الدالة بقراءة او قيمتين من obs1 وعندما وجدت انها اكتملت لم تقرأ بقية القيم وانما انتقلت إلى obs2 وقامت بقراءة قيمة واحدة منه وعندما وجدت خطأ انتقلت إلى obs3 وقامت بقراءة جميع القيم ولكن لم تنتقل إلى obs4 لأن obs3 لم يكتمل او يحدث فيه خطأ، ونستفيد من هذه الدالة في حالة مثلاً كنا نقرا بيانات من أكثر من مصدر source مثلاً لدينا موقع نعرض فيه بيانات الطقس ونأخذ هذه البيانات من أكثر من API، ونُريد في حالة وجود خطأ في أحد API ان يقوم بقراءة بيانات الطقس من API التالي بحيث لا يتعطل الموقع لدينا، طبعاً هذا كمثال ولكن قد يكون هنالك استخدامات كثيرة بحسب الاحتياج.

3- Pipeable Operators:

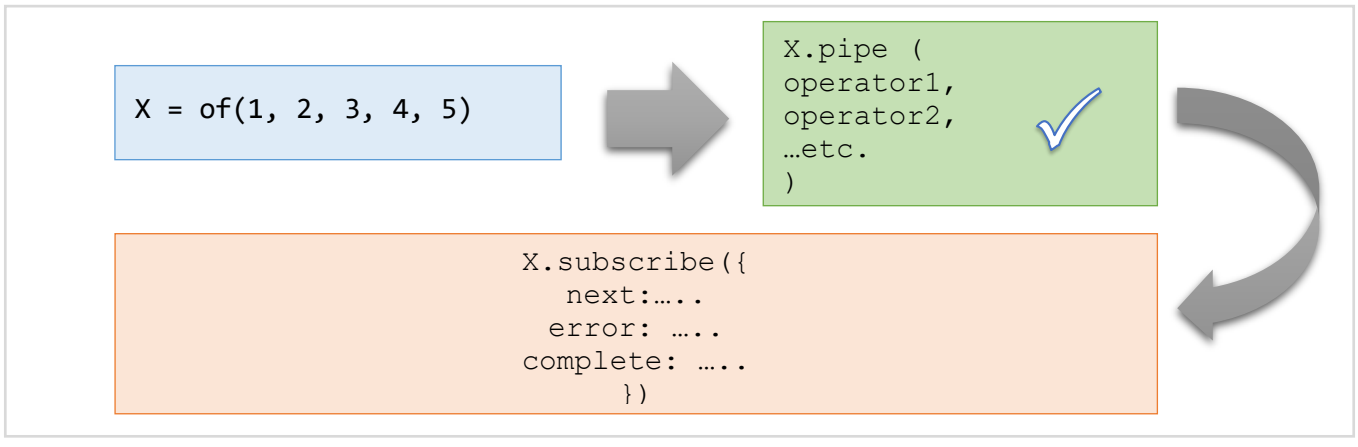
بعدما تكلمنا في القسم السابق عن أغلب دوال Static مع الشرح والتوضيح وضرب الأمثلة، سوف نتكلم في هذا الجزء عن القسم الثاني من هذه الدوال وهي دوال pipe، وهذه الدوال لا تقل أهمية عن سابقتها فجميع هذه الدوال هي مكملة لبعضها البعض فقد تحتاج عزيزي المتعلم إلى المزج والدمج بين جميع هذه الدوال للوصول للبيانات التي تناسب احتياجك.

أما من ناحية وظيفة دوال pipe فهي دوال يتم جمعها جميعا بشكل متتابع او متسلسل في دالة pipe، لتلاعب والتحكم في البيانات القادمة لنا على شكل Observable قبل لا نعمل لها subscription والرجاء الانتباه إلى هذه النقطة جيداً فمهمة دوال pipe تحدث بعدما يتم عمل emit للقيم وقبل ان يتم عمل لها subscription، في منطقة معينة في الوسط.

اما من ناحية ما الذي تقدمه لنا هذه الدوال فهي تقدم خيارات كثيرة منها التعديل على هذه البيانات وايضاً نستطيع تصفية وفلترة البيانات واجراء العمليات الحسابية عليها او دمجها في اكثر من Observable إذا كانت هذه البيانات متعددة المصادر sources وايضاً نستطيع تحويل Observable من Multicast إلى Unicast والعكس ايضاً، ونستطيع التحويل من Hot Observable إلى Cold Observable والعكس صحيح، ومن الأمور المهمة نستطيع التعامل مع Observables المتداخلة أو ما يسمى Higher Order Observable، وغيرها الكثير من الأمور التي تُساعدنا كمطورين بالتحكم والتلاعب بالبيانات بما يلبي احتياجنا.

وطريقة استخدامها بشكل عام بسيط حيث نكتب الدالة الرئيسية pipe وبين قوسين نبدأ بإضافة هذه الدوال وكل دالة تنتهي من الإجراءات الخاصة بها تعمل return للبيانات لدالة التي تليها وهكذا إلى ان تنتهي جميع الدوال، وكما ان لكل دالة طريقة معينة لكتابتها ولكن بشكل عام تتشابه أغلبها بطريقة الاستخدام وتختلف بالوظيفة والفائدة منها.

ونستطيع تمثيل طريقة استخدام دوال pipe مع Observer وSubscriber من خلال الشكل التالي:



ومن ناحية الاستدعاء لهذه الدوال فتتم بالطريقة التالية:

import {..., name of operator, ...etc.} from "rxjs";

import {..., name of operator, ...etc.} from "rxjs/operators";

وأخيراً يجب الإشارة انه ممكن ان نستخدم دوال static بداخل دوال pipe والعكس صحيح، وهو تأكيد لما قلته في بداية كلامي عن هذه الدوال ان جميع الدوال هي في الحقيقة مكملة لبعضها البعض وليست منفصلة، بمعنى لا يعني عند استخدامنا لدوال pipe ان نستغني عن دوال static.

وسوف نستعرض أغلب هذه الدوال بالشرح والتوضيح وإعطاء الأمثلة من خلال تجميع الدوال المتشابهة مع بعضها البعض بشكل مجموعات، كالتالي: (المصدر RxJS - RxJS Operators (rxjs-dev.firebaseapp.com))

1. Transformation Operators	2. Filtering Operators
3. Join Operators	4. Multicasting Operators
5. Utility Operators	6. Error Handling Operators
7. Conditional and Boolean Operators	8. Mathematical and Aggregate Operators

1-3- Transformation Operators:

هذه اول مجموعة من دوال pipe حيث تقوم هذه الدوال بإجراء عمليات متعددة على Observable كتخزين البيانات مؤقتاً في حال انقطاع الاتصال مع Subscriber الخاص بعرض البيانات في التطبيق ومن ثم إعادة ارسال هذه البيانات إلى Subscriber عند رجوع الاتصال، ومنها دمج أكثر من Observable مع بعضها البعض او تجزئة Observable او اقتطاع جزء معين من أي Observable، او تحويل مسار هذا Observable، كأن يكون لدينا Observable يجلب أرقام المعلمين، فنستطيع تغيير البيانات في هذا Observable كأن نقوم مثلاً بجلب بيانات الطلاب الذي يقوم بتدريسهم المعلم المحدد، وبذلك قمنا بتغيير البيانات من بيانات المعلمين إلى بيانات الطلاب، في الحقيقة هنالك الكثير والكثير من هذه الدوال والتي تستطيع الاطلاع عليها جميعاً من خلال الموقع الرسمي لهذه المكتبة.

وبطبيعة الحال من الصعوبة عرض جميع هذه الدوال التي تنتمي إلى هذا المجموعة، ولكن سوف نستعرض أغلبها، وتجب الإشارة انه اهم هذه الدوال والتي يجب على مستخدمي اطار عمل Angular اتقانها هي دوال Mapping و Higher Order Mapping وتأتي بعدها بالأهمية scan وgroupBy وtoArray وpairwise، اما باقي الدوال فهي مهمة ولكن احتياجنا لها كمطوري Angular قليل لذلك شرحتها وأوردت الأمثلة عليها لكي تكون عزيزي المتعلم على اطلاع بأغلب هذه الدوال لذلك لا ضير ان تطلع عليها وتحاول فهمها لأنك قد تحتاج لها في عملك.

1-1-3- map()/mapTo() Operators:

سوف نبدأ بالدالة map في البداية، وهي مشابهه نوعاً ما بالدالة map الموجودة ضمن المصفوفات الخاصة بالجافا سكريبت، حيث تقوم باستقبال مُدخل وهو Observable (Stream of Data) ومن ثم تُعيد Observable آخر سواء يحتوي على نفس القيم او يقوم بتغيير هذه القيم بشكل جزئي او بشكل كامل.

وتعتبر هذه الدالة من أهم دوال pipe وأكثرها استخداماً والتي يجب ان يُجيدها أي مطور يُريد تعلم مكتبة Rxjs.

وفي العادة يتم استخدام هذه الدالة لتعديل في قيم Observable سواء اقتطاع جزء مُعين من هذا Observable او ارجاع قيم أخرى تختلف عن القيم الموجودة في Observable المُدخل.

ولتوضيح لنبدأ بمثال بسيط ومن ثم نتدرج في الصعوبة، والمثال الأول هو بكل بساطة نريد التلاعب بقيم Observable معين بحيث نجري عليه عملية حسابية بسيطة وهي جمع القيمة ونفسها، لذلك لنفرض ان لدينا Observable التالي:

ملف app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  ngOnInit(): void {
    of(1, 2, 3, 4, 5).subscribe(console.log);
  }
}
```

ولو شاهدنا النتيجة سنجد ان القيم سيتم عرضها كما هي، ولكن كما قلنا سابقاً نحتاج إلى تغيير هذه القيم بحيث نجمع القيمة ونفسها أي 1 + 1 و2 + 2 وهكذا بقية القيم.

وبما أننا نريد تغيير وتعديل القيم ففي هذه الحالة نستخدم الدالة map، وبما ان الدالة map من دوال pipe فنستخدمها قبل subscription أي القيم تصل إلى subscriber وهي بشكلها النهائي بعد التعديل، كالتالي:

ملف app.component.ts

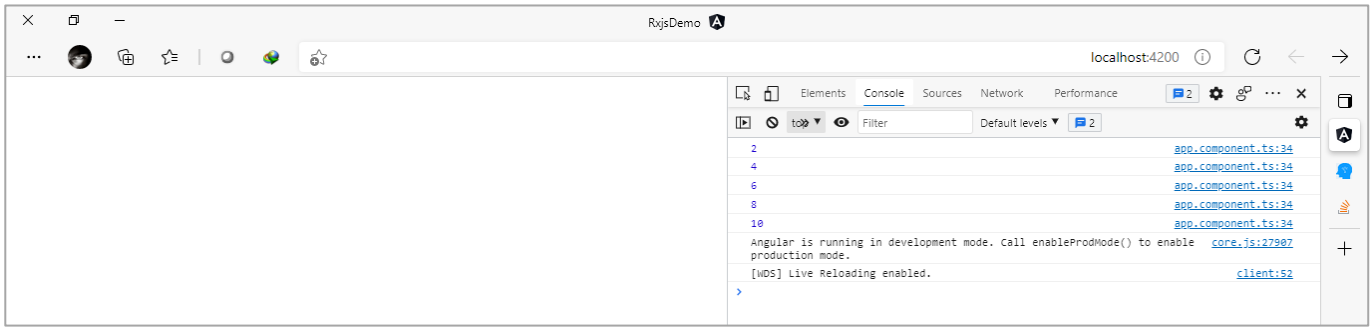
```
import { Component } from '@angular/core';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  ngOnInit(): void {
    of(1, 2, 3, 4, 5)
      .pipe(
        map((value: number) => {
          return value + value;
        })
      )
      .subscribe((value) => console.log(value));
  }
}
```

نلاحظ وضعنا دوال pipe بعد Observable وقبل Subscription، وفي حالتنا هذه لا توجد إلا دالة واحدة وهي map حيث هي الأخرى تستقبل دالة والبارامتر الخاص بهذه الدالة هو عبارة عن القيم التي تم عمل emit لها من Observable الأساس (وقد تكون هذه القيم هي عبارة عن قيم تم عمل return من دالة سابقة)، وهذه الدالة قامت بالمرور على جميع القيم وفي كل مرة تقوم بجمع القيمة ونفسها ومن ثم تعمل لها return، ولو كان هنالك دالة أخرى فإن ناتج هذه الدالة يُعتبر مدخل للدالة التالية وهكذا إلى ان تنتهي جميع دوال pipe. وأخيراً عملنا subscribe لكي نقرأ هذه القيم.

وقبل ان نشاهد النتيجة في المتصفح، لنقوم باختصار الاسطر السابقة باستخدام ميزات ES6، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  ngOnInit(): void {
    of(1, 2, 3, 4, 5)
      .pipe(map((value: number) => value + value))
      .subscribe((value) => console.log(value));
  }
}
```

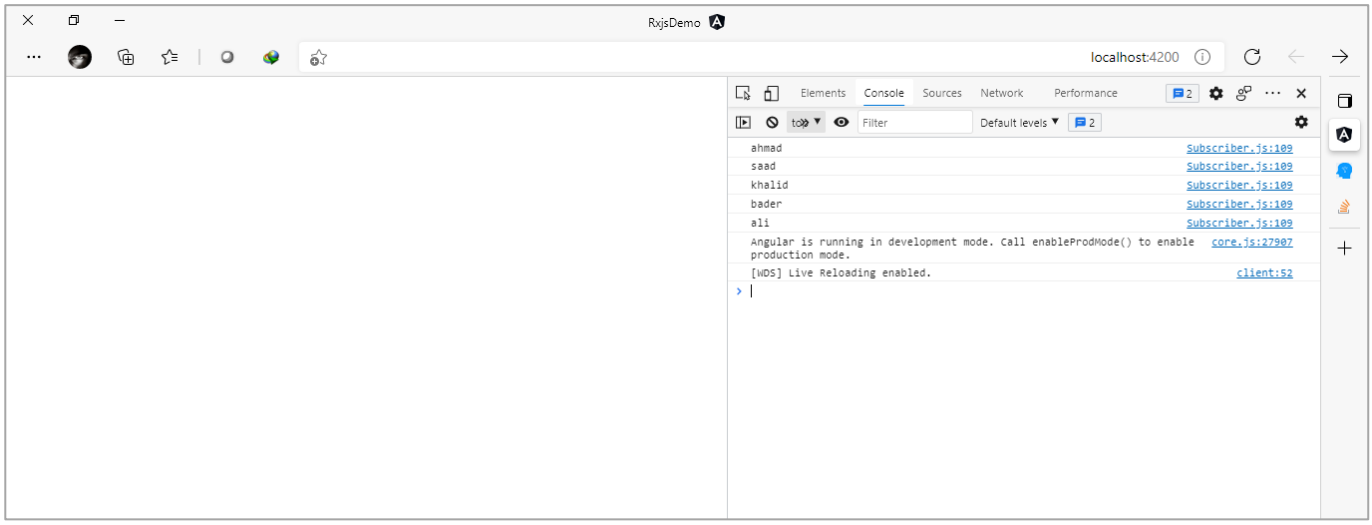


اما في المثال التالي لنفرض ان لدينا مجموعة من البيانات لمجموعة من الطلاب نُريد ان نستخرج فقط أسماء الطلاب ونتجاهل بقية البيانات، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { from, of } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  private students = from([
    { id: 1, name: 'ahmad', city: 'Riyadh', mobile: 5555555555 },
    { id: 2, name: 'saad', city: 'Dammam', mobile: 5555444555 },
    { id: 3, name: 'khalid', city: 'Jeddah', mobile: 5555555111 },
    { id: 4, name: 'bader', city: 'Makkah', mobile: 5500005555 },
    { id: 5, name: 'ali', city: 'ALKhobar', mobile: 5555558888 },
  ]);
  ngOnInit(): void {
    this.students
      .pipe(map((students: { name: string }) => students.name))
      .subscribe(console.log);
  }
}
```



نلاحظ استقبلنا Observable يحتوي على مجموعة من البيانات ومن ثم استخدمنا الدالة map لكي نقرأ فقط الاسم الخاص بكل طالب.

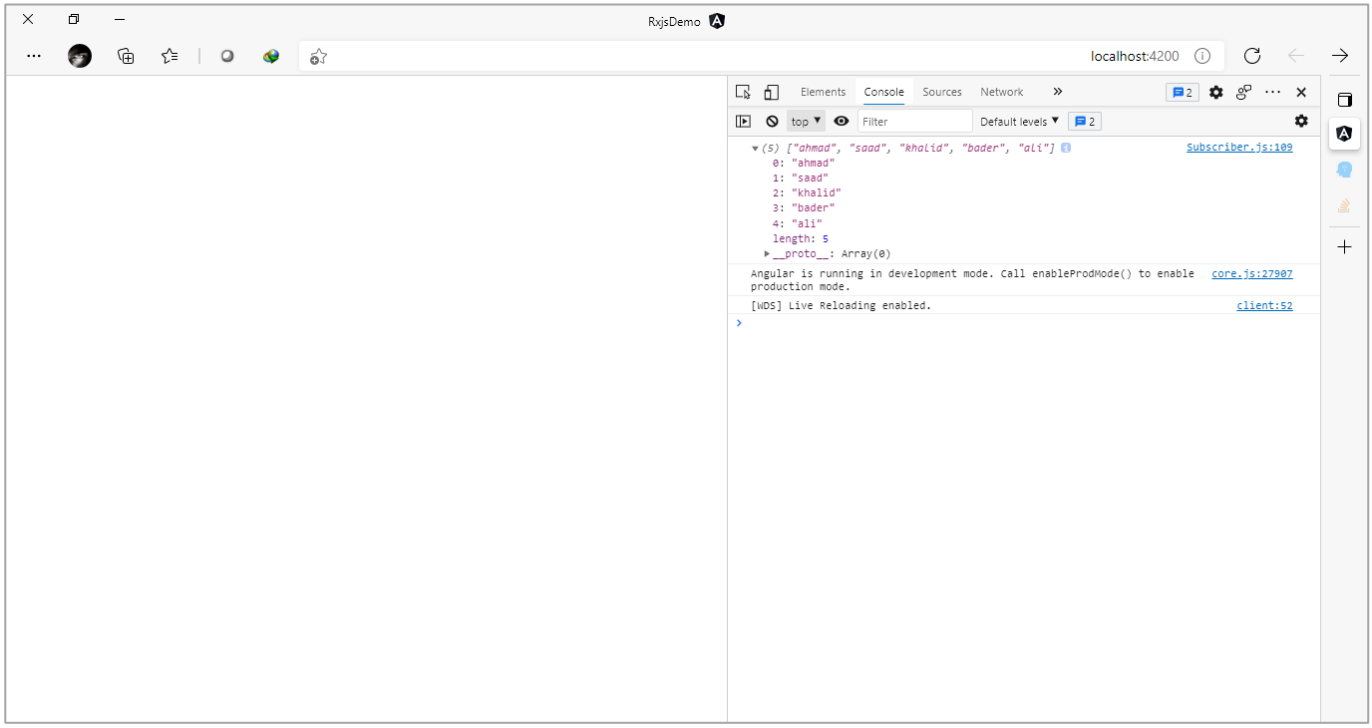
اما المثال الآخر، لنزيد الجرعة قليلاً، في المثال السابق استخدمنا الدالة from وكما هو معلوم ان هذه الدالة تقوم بعمل flatting للمصفوفة، ولكن ماذا لو استخدمنا الدالة of لكي تم عمل emit للمصفوفة كاملة، ففي هذه الحالة كيف نستخدم الدالة map للحصول على الخاصية name فقط؟

ويتم هذا الامر في البداية باستخدام الدالة map للحصول على المصفوفة ومن ثم نستخدم الدالة map الخاصة بالمصفوفات لقراءة عناصر المصفوفة كاملة ومن ثم عمل ارجاع return للخاصية name على شكل مصفوفة، والهدف من هذا المثال لتأكيد على ما قيل سابقاً على ان عمل الدالة map الخاصة بالObservable هي مشابهة نوعاً ما للدالة map الخاصة بالمصفوفات، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { from, of } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {

  public students = of([
    { id: 1, name: 'ahmad', city: 'Riyadh', mobile: 5555555555 },
    { id: 2, name: 'saad', city: 'Dammam', mobile: 5555444555 },
    { id: 3, name: 'khalid', city: 'Jeddah', mobile: 5555555111 },
    { id: 4, name: 'bader', city: 'Makkah', mobile: 5500005555 },
    { id: 5, name: 'ali', city: 'ALKhobar', mobile: 5555558888 },
  ]);

  ngOnInit(): void {
    this.students
      .pipe(
        map((students: any[]) => {
          return students.map((student) => {
            return student.name;
          });
        })
      )
      .subscribe(console.log);
  }
}
```

والنتيجة، كالتالي:



وفي المثال التالي نُجيب عن ماذا إذا احتجنا إلى أكثر من map بنفس دالة pipe؟

والإجابة بسيطة نضع كل map وراء الأخرى بشكل متسلسل chain، بحيث عند انتهاء الأولى من عملها تُعيد قيمة لدالة الmap التي تليها وهذه القيمة تعتبر مدخل لدالة التي تلي الmap الأولى وهكذا إلا ان ننتهي من جميع maps.

ولتوضيح لنفرض انه لدينا قائمة منسدلة تعرض أسماء الطلاب وعند اختيار اسم من هذه الأسماء نُريد ان نعرض المدينة التي يوجد بها هذا الطالب.

لذلك اول شيء نقوم به هو تصميم القائمة وملئها بالقيم، كالتالي:

ملف app.component.html

```
<select (change)="getChild($event.target.value)">
  <option ngValue=""></option>
  <option *ngFor="let student of students | async">
    {{ student.name }}
  </option>
</select>
```

وفي ملف class لهذا component نكتب الشفرة البرمجية التالية:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';
export interface Students {
  id: number;
  name: string;
  city: string;
  mobile: number;
```

```
  parentId: number;
}
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  public students = of([
    { id: 1, name: 'ahmad', city: 'Riyadh', mobile: 5555555555 },
    { id: 2, name: 'saad', city: 'Dammam', mobile: 5555444555 },
    { id: 3, name: 'khalid', city: 'Jeddah', mobile: 5555555111 },
    { id: 4, name: 'bader', city: 'Makkah', mobile: 5500005555 },
    { id: 5, name: 'ali', city: 'ALKhobar', mobile: 5555558888 },
  ]);
  getChild(value: string): void {
    this.students
      .pipe(
        map((students: Students[]) => {
          return students.filter((student) => {
            return student.name === value;
          });
        }),
        map((students: Students[]) => {
          return students.map((student) => {
            return student.city;
          });
        })
      )
      .subscribe(console.log);
  }
}
```

الأولى

الثانية

نلاحظ انه في map الأولى استخلصنا جميع بيانات الطالب المحدد، ومن ثم عملنا له return وفي الmap الثانية استقبلنا هذه البيانات واستخرجنا منها فقط اسم المدينة ومن ثم عملنا return لهذه القيمة وبما انه لا يوجد دالة أخرى فإن هذه القيمة ذهبت مباشرة إلى Subscriber لعرض هذه القيمة.

وتجدر الاشارة ان الحل السابق ليس أفضل الحلول وخصوصاً لمن أراد ان يستخدم فقط دوال Rxjs بدون استخدام أي من دوال المصفوفات، ولكن هذه الحلول تتطلب منا ان نستعرض دوال أخرى وهو ما سوف نفعله لاحقاً **حيث سنعيد كتابة هذا المثال ولكن بالاعتماد بشكل كامل على دوال Rxjs.**

ولننتقل إلى المثال التالي، والذي سوف نشرح فيه كيفية استخدام دوال maps المتداخلة، حيث تقوم فكرة المثال انه لدينا أثنين Observables الأول يمثل بيانات أولياء الأمور والثاني يمثل بيانات الطلاب ويوجد مع بيانات كل طالب رقم ولي الأمر الذي يتبعه هذا الطالب، والنتيجة النهائية هي وجود قائمة منسدلة تحتوي على أسماء أولياء الأمور وعند اختيار أحد الأسماء تظهر لنا أسماء الطلاب التابعين لاسم ولي الأمر المُختار.

وبطبيعة الحال يمكن حله بأكثر من طريقة وهذا ما يجعل للبرمجة متعة، وهنا حالياً سوف استخدم طريقتين لحل هذا المثال، **ولاحقاً عند تعلم دوال جديدة سوف نعيد حل هذا المثال بطرق أفضل و أكثر احترافية.**

اما الآن لنضيف Markup اللازم في ملف HTML، كالتالي:

ملف app.component.html

```
<select (change)="getChild($event.target.value)">
  <option ngValue=""></option>
  <option *ngFor="let parent of parents | async">
    {{ parent.name }}
  </option>
</select>
<ul>
  <li *ngFor="let studentName of studentsName">{{ studentName }}</li>
</ul>
```

نلاحظ هنا زدنا درجة الصعوبة قليلاً حيث لم نُضف رقم ولي الأمر في أداة option عن طريق ngValue كما فعلنا في المثال السابق، والسبب اننا نريد الوصول إلى رقم ولي الأمر عن طريق البحث في Observable اعتماداً على اسم ولي الامر.

اما الآن لنستعرض Logic البرمجي في ملف class لهذا component، ومن ثم نقوم بشرح اهم اجزاءه، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

export interface Students {
  id: number;
  name: string;
  city: string;
  mobile: number;
  parentId: number;
}
export interface Parents {
  id: number;
  name: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {

  studentsName: string[];

  public parents = of([
    { name: 'Faisal', id: 111 },
```

```
    { name: 'Fahd', id: 222 },
    { name: 'Mohammed', id: 333 },
    { name: 'Sultan', id: 444 },
  ]);
  public students = of([
    { id: 1, name: 'ahmad', city: 'Riyadh', mobile: 5555555555, parentId: 111 },
    { id: 2, name: 'saad', city: 'Dammam', mobile: 5555444555, parentId: 111 },
    { id: 3, name: 'khalid', city: 'Jeddah', mobile: 5555555111, parentId: 222 },
    { id: 4, name: 'bader', city: 'Makkah', mobile: 5500005555, parentId: 333 },
    { id: 5, name: 'ali', city: 'ALKhobar', mobile: 5555558888, parentId: 444 },
  ]);

  getChild(name: string): void {
    this.parents
      .pipe(
        map((parents: Parents[]) => {
          return parents.find((parent) => {
            return parent.name === name;
          });
        }),
        map((parent: Parents) => {
          return parent.id;
        }),
        map((id: number) => {
          return this.students
            .pipe(
              map((students: Students[]) => {
                return students.filter((student) => {
                  return student.parentId === id;
                });
              }),
              map((students: Students[]) => {
                return students.map((student: Students) => {
                  return student.name;
                });
              })
            )
            .subscribe((names) => (this.studentsName = names));
        })
      )
      .subscribe();
  }
}
```

الأولى

الثانية

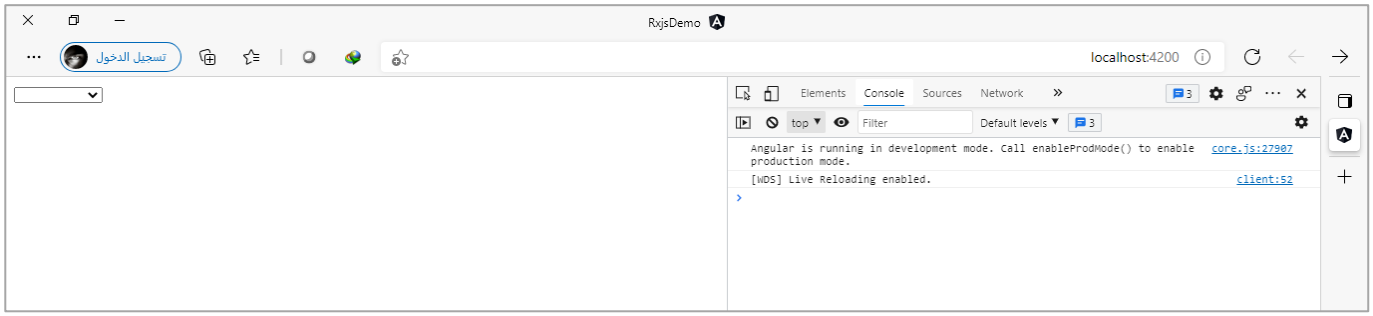
الأولى

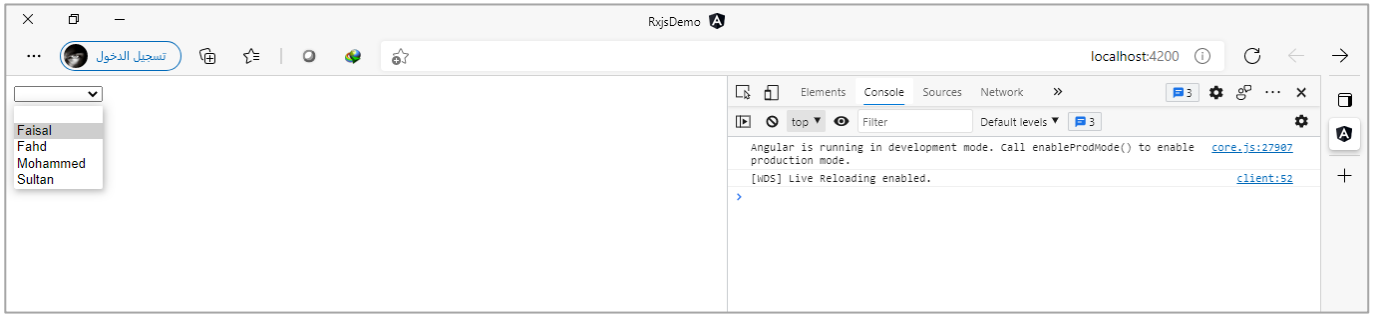
الثانية

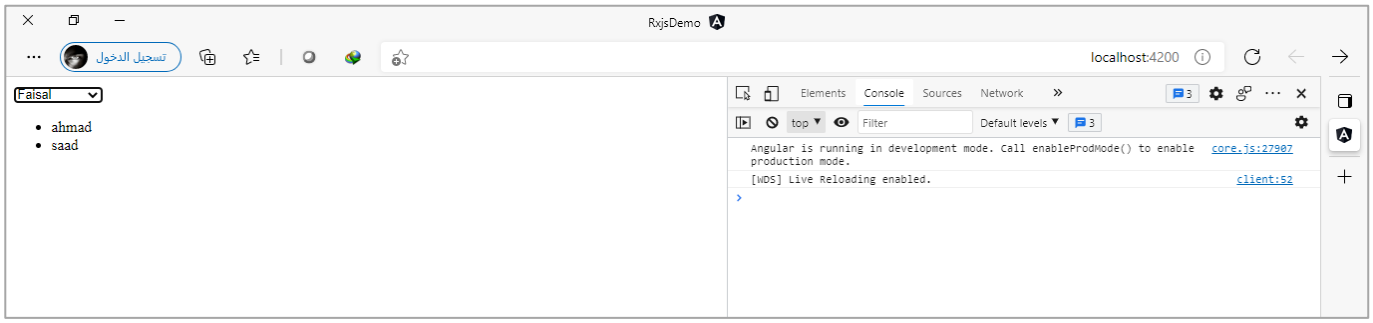
الثالثة

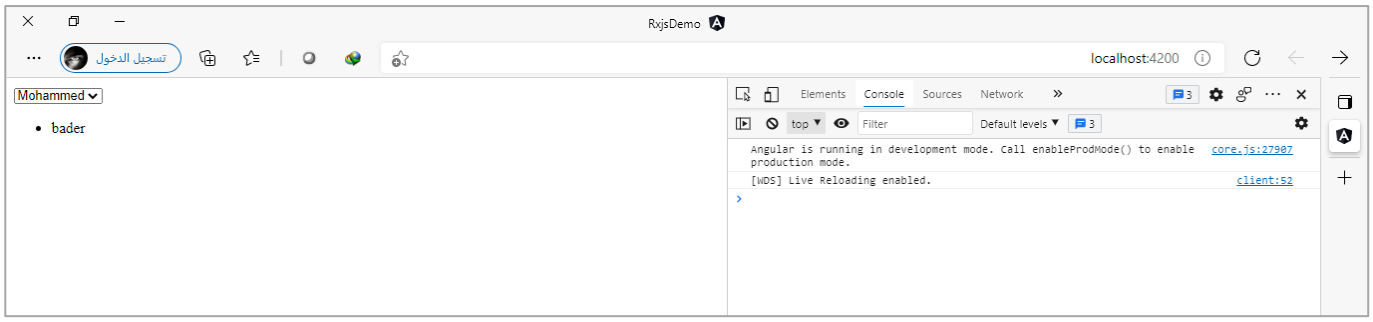
نلاحظ لكي نقوم بحل المثال اعتمدت فقط على الدالة map (أُعيد وأكرر لتأكيد ان هذا الحل ليس الحل الأفضل وانما اخترته هنا لأننا نُريد ان نركز على الدالة map فقط وعندما نتعلم بعض الدوال الأخرى سنعيد بناء هذا المثال بطريقة أفضل بإذن الله) بشكل متسلسل chain ومتداخل nested، حيث الأولى لكي نتحصل على بيانات ولي الأمر المتوافقة مع الاسم المختار من القائمة المنسدلة، ومن ثم عملت return لهذه القيمة، اما الثانية فتستقبل المدخل القادم من map الأولى ومن ثم نستخلص من هذه البيانات رقم ولي الامر ونعم لها return، وفي map الثالثة نستقبل رقم ولي الامر في متغير

اسميته id، ومن ثم استدعينا الObservable الثاني ذو الاسم students لكي نتحصل على اسم الطلاب منه بناءً على رقم ولي الامر، وللقيام بذلك احتجنا إلى اثنين maps الأولى لكي تستخرج لنا بيانات الطلاب الذين رقم ولي الامر في بياناتهم مساوٍ لرقم ولي الامر id، والثانية لاستخراج اسم الطلاب فقط وتجاهل بقية البيانات، وأخيراً هذه الأسماء اضفناها في المصفوفة studentsName لكي نعرضها للمستخدم، والآن لنشاهد النتيجة في المتصفح، كالتالي:









وسوف تلاحظ عزيزي المتعلم أن هذا الحل ليس جيداً لأنه يعتبر من Higher Order Observable أي مجموعة من الObservables المتداخلة، وقد أشرنا سابقاً ان هذا النوع نحتاج إلى Operators خاصة لتعامل معه، وهو ما سوف نتكلم عنه بإذن الله في الجزء التالي.

أما الآن لنستعرض طريقة أخرى لحل هذا المثال عن طريق استخدام إحدى طرق دمج Observables مع بعضها البعض والتي تكلمنا عنها سابقاً، وفي مثالنا هذا لك حرية اختيار أي طريقة تُريدها سواء كانت zip او combineLatest او forkJoin

لأننا سنستخدم الدالة of لتحويل القيم إلى Observable وكما هو معروف ان هذه الدالة سوف تعمل emit لجميع قيمها ومن ثم تغلق هذا Observable وهذا معناه ان أي دالة من هذه الدوال سوف تقرأ هذا Observable بنفس الطريقة وسوف تُعطي نفس النتائج.

وأنا سوف استخدم الدالة combineLatest، في هذا المثال مع الاستفادة من ميزات ES6 لتسهيل وتبسيط واختصار الاسطر البرمجية، كالتالي:

ملف app.component.ts

```
import { Component } from '@angular/core';
import { combineLatest, of } from 'rxjs';
import { map } from 'rxjs/operators';

export interface Students {
  id: number;
  name: string;
  city: string;
  mobile: number;
  parentId: number;
}
export interface Parents {
  id: number;
  name: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent {
  studentsName: string[];
  public parents: Observable<Parents[]> = of([
    { name: 'Faisal', id: 111 },
    { name: 'Fahd', id: 222 },
    { name: 'Mohammed', id: 333 },
    { name: 'Sultan', id: 444 },
  ]);
  public students: Observable<Students[]> = of([
    { id: 1, name: 'ahmad', city: 'Riyadh', mobile: 5555555555, parentId: 111 },
    { id: 2, name: 'saad', city: 'Dammam', mobile: 5555444555, parentId: 111 },
    { id: 3, name: 'khalid', city: 'Jeddah', mobile: 5555555111, parentId: 222},
    { id: 4, name: 'bader', city: 'Makkah', mobile: 5500005555, parentId: 333 },
    { id: 5, name: 'ali', city: 'ALKhobar', mobile: 5555558888, parentId: 444 },
  ]);

  getChild(name: string): void {
    combineLatest([this.parents, this.students])
      .pipe(
```

```
        map(([parents, students]) => {
          const id = parents.find((parent) => parent.name === name).id;
          return { id, students };
        }),
        map(({ id, students }) => {
          const student = students.filter((_students) => _students.parentId === id);
          return student.map((_student) => _student.name);
        })
      )
      .subscribe((names) => (this.studentsName = names));
  }

}
```

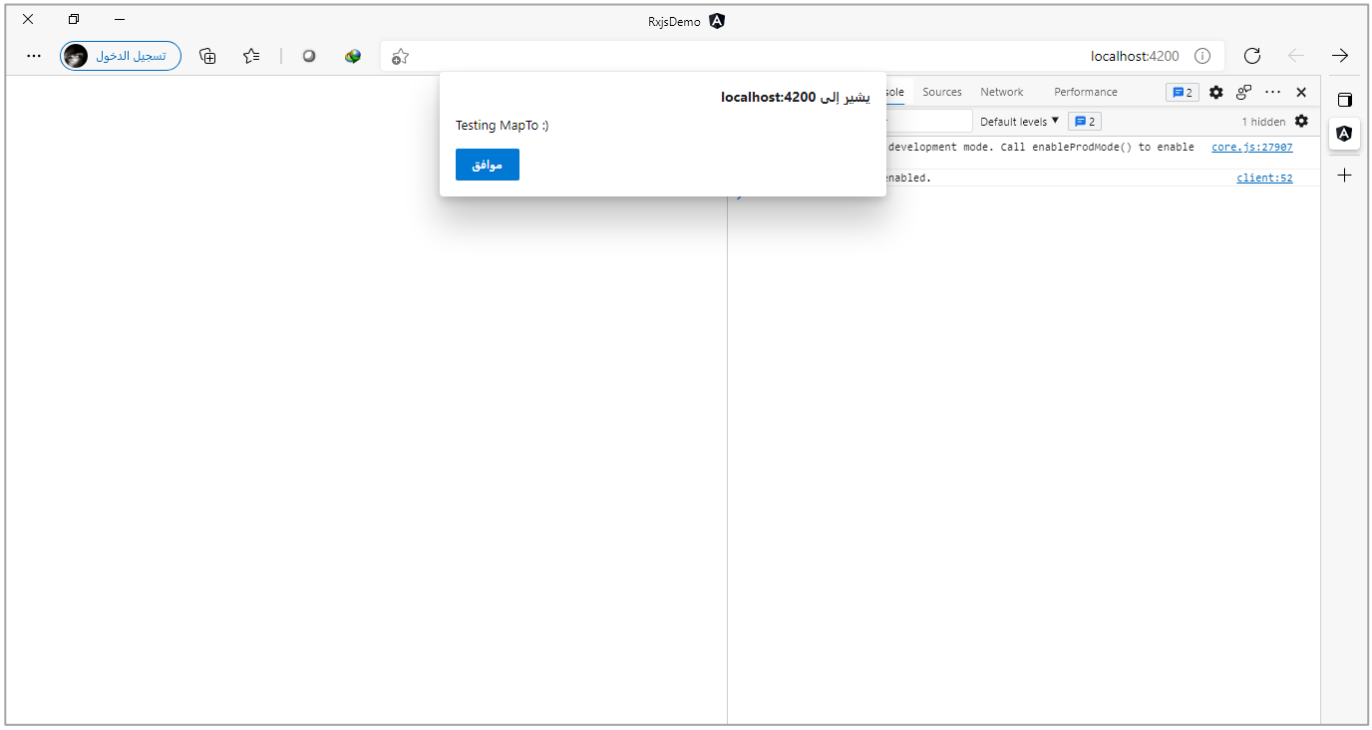
ولو شاهدنا النتائج فسوف نلاحظ نفسها سوف تظهر لنا، ولعل هذه الطريقة أفضل من الطريقة الأولى، والسبب من استعراضي للطريقة الأولى هو لكي تعرف عزيزي المتعلم كيف تتعامل مع maps المتداخلة، وايضاً لتهيئتك ذهنياً للجزء التالي وهي دوال Higher Order Observable.

اما بخصوص الدالة الأخرى وهي دالة mapTo، فأمرها سهل وبسيط فهي تقوم بتحويل قيمة Observable إلى قيمة ثابتة، فمثلاً لو تم الضغط بزر الفأرة الأيسر على أي مكان في صفحة المتصفح، نُريد ان نظهر رسالة للمستخدم، كالتالي:

app.component.ts ملف

```
import { Component } from '@angular/core';
import { fromEvent} from 'rxjs';
import { mapTo } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {
  ngOnInit(): void {
    fromEvent(document, 'click')
      .pipe(mapTo('Testing MapTo :)'))
      .subscribe((value) => alert(value));
  }
}
```

والنتيجة في المتصفح، تكون كالتالي:



3-1-2- Higher Order Mapping Operators:

والمقصود بها الدوال التالية:

- mergeAll() - mergeMap()
- concatAll() - concatMap()
- switchAll() - switchMap()
- exhaustAll() - exhaustMap()

الدالتين map وmapTo تُسمى First Order أي انها تقوم باستقبال Observable ومن ثم تعمل return لقيمة أخرى.

ولكن ماذا لو احتجنا إلى ان نتعامل مع مجموعة من Observables المتداخلة، كما في المثال السابق حيث استقبلنا Observable يحتوي على قيمة رقم ولي الأمر ومن ثم في map التالية احتجنا إلى Observable جديد، فهذه الحالة تُسمى Higher Order Observable وقد تكلمنا عنها سابقاً، وهذا النوع من Observable لا تنفع معه الدالة map وانما نحتاج إلى مجموعة من الدوال الخاصة والتي تُسمى Higher Order Mapping، حيث جميعها تشترك بانها تستقبل Observable خارجي outer ومن ثم تعمل subscribe بشكل تلقائي للObservable الداخلي والذي يسمى inner وتقرأ قيمة ومن ثم تُعيد هذه القيم للOuter Observable وتعمل unsubscribe بشكل تلقائي للInner Observable.

وايضاً تشترك جميعها وظيفة أخرى وهي انها تعمل flatten للObservable إذا كان على شكل مصفوفة، بحيث تقوم بتحويل قيم هذا الObservable من مصفوفة array إلى قيم منفردة.

هذا من ناحية التشابه، اما من ناحية الاختلافات بينها فهي تختلف في طريقة التعامل وقراءة القيم بين هذه Observables المختلفة، كما كنا نقول سابقاً في دوال الدمج، حيث أشرت ان هذه الدوال تشترك جميعها في انها تدمج أكثر من Observable ولكن تختلف في طريقة وآلية هذا الدمج، وهنا ايضاً تقريباً الأمر متشابهة حيث جميعها تتعامل مع Observables المتداخلة ولكن تختلف في آلية وطريقة قراءة هذه القيم.

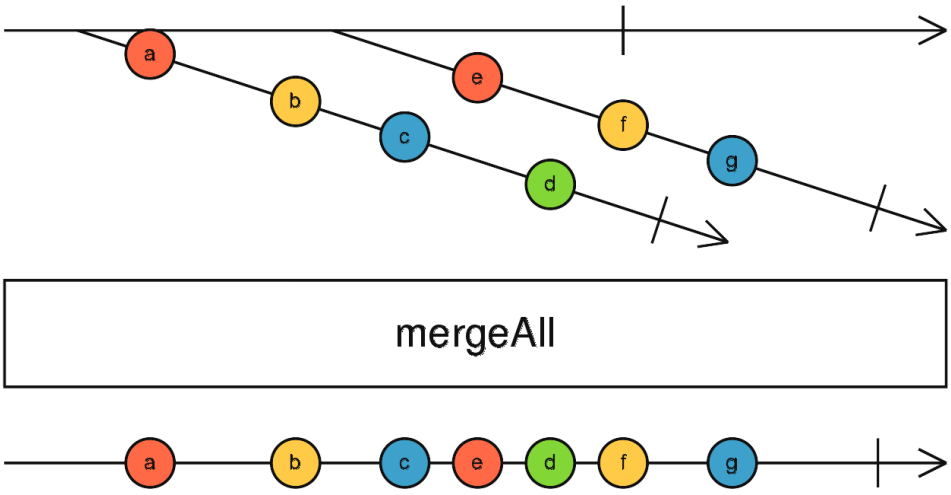
اما الآن لنقوم باستعراض هذه الدوال بشيء من التفصيل مع الشرح بالأمثلة.

3-1-2-1- mergeAll() /mergeMap():

هذه الدالة تقريباً مشابهة لدالة merge التي تكلمنا عنها سابقاً حيث تقوم بعمل دمج لكلا الObservables الخارجي outer والداخلي inner في Observable واحد، وايضاً هي لا تهتم بالترتيب فأي قيمة تصل اولاً من أي Observable هي التي يتم قراءتها، وهي بذلك مشابهة للدالة merge.

ولنبدأ بأول دالة وهي mergeAll، حيث تقوم هذه الدالة بتطبيق مفاهيم merge التي تكلمنا عنها سابقاً ولكن هنا على Observables المتداخلة حيث تستقبل الOuter Observable ومن ثم تعمل له دمج مع Inner Observable وأول قيمة يتم لها emit من كلا الObservables هي التي يتم قراءتها وهكذا إلى ان تنتهي من جميع القيم.

وايضاً هي تتعامل Higher Order Observables حيث تعمل subscribe بشكل تلقائي لInner Observable لكي تقرأ القيم منه ومن ثم تعمل unsubscribe بشكل تلقائي عند الانتهاء.



نلاحظ من الشكل السابق يوجد لدينا اثنين Observables متداخلة حيث الObservable الأول او ما يسمى Outer Observable يحمل القيم من a إلى d، اما Inner Observable فإنه يحمل القيم من e إلى g، لذلك عند استخدام الدالة mergeAll ستقوم بعمل دمج لكلا الObservables بعدما تعمل flatten – (تقرأ القيم من Higher Order Observables) – ولكن طريقة القراءة مشابهة لدالة merge حيث اول قيمة يتم عمل لها emit هي التي يتم قراءتها.

ولتوضيح لنقوم بإعادة المثال السابق والخاص بالدالة merge مع بعض التعديلات البسيطة لكي يتناسب مع المفاهيم

التي سوف نطرحها هنا، كالتالي:

ملف app.component.html

```
<div>
  <img *ngFor="let image of images" [src]="image" />
</div>
```

ملف app.component.scss

```
div {
  box-sizing: border-box;
  width: 300px;
  margin: 0 auto;
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
  align-items: center;
}
img {
  margin: 20px;
  border: 5px solid rgb(145, 145, 145);
}
```

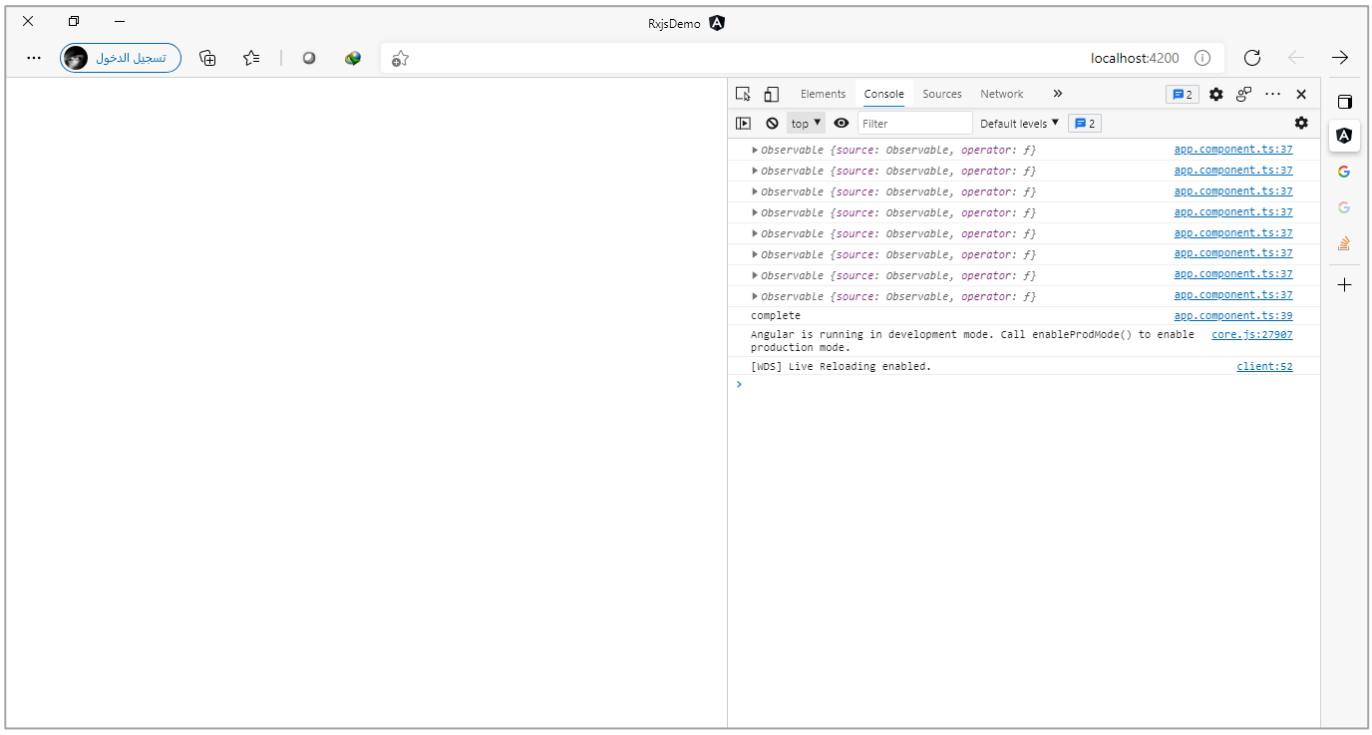
جميع ما سبق لم اقم بأي تعديل عليه والتعديل الذي سوف يحدث هو في ملف class، حيث سنقوم بإنشاء اثنين Observable الأول هو عبارة عن مجموعة من الأسماء (التصنيفات) لأماكن وحيوانات و ...الخ، والثاني هو عبارة عن الرابط الذي نريد ان ندمج هذه الأسماء فيه، بحيث يكون لدينا أثنين Observables الأول وهو Outer Observable يُمثل التصنيفات ومن ثم نعمل له map ونستدعي الObservable الثاني وهو Inner Observable والذي يمثل الرابط URL والذي سوف يحمل في كل مرة تصنيف من هذه التصنيفات لكي يجلب لنا صورة عشوائية من نفس هذا التصميم، وفي البداية لن نستعمل mergeAll وانما سنستخدم map العادية ولنرى النتيجة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  public images: string[] = [];

  private categories=of('all', 'brazil', 'rio', 'paris', 'tree', 'saudi', 'hero','man');
  private imgUrl = of('https://loremflickr.com/100/100');
```

```
  ngOnInit(): void {

    this.categories
      .pipe(                                          Outer Observable
        map((category) => {
          return this.imgUrl.pipe(
            map((url) => {                            Inner Observable
              return url + '/' + category;
            })
          );
        })
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('complete'),
      });
  }
}
```

كما هو واضح يوجد لدينا أثنين Observables وهي حالة Higher Order Observable التي اشبعناها شرحاً وتفصيلاً، ولنقم الآن بتشغيل التطبيق في المتصفح ولنرى النتيجة، كالتالي:



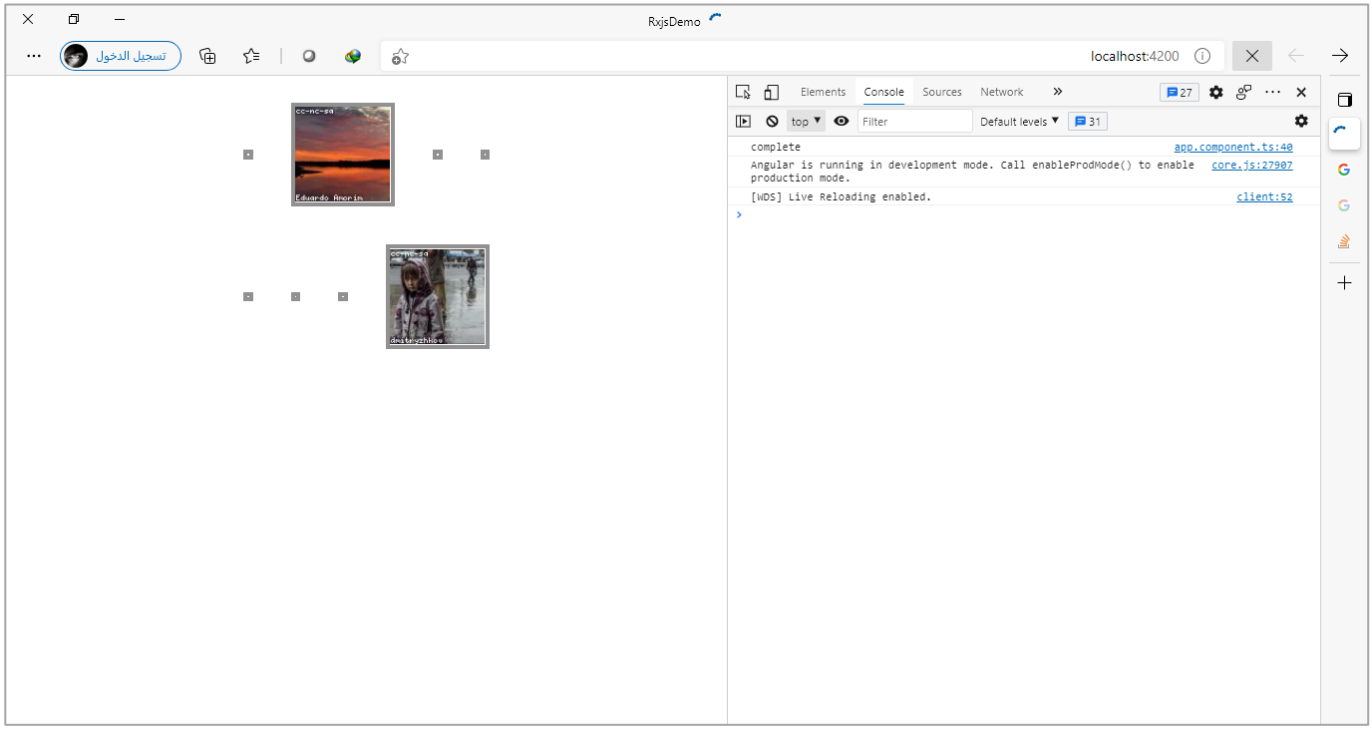
نلاحظ النتيجة كما هو متوقع عرض لنا الInner Observable ولم يعرض القيم لأننا لم نعمل له Subscribe، لذلك اما ان نعمل subscribe لهذا الObservable او ان نستخدم أحدى دوال Higher Order Mapping، وهو الذي سوف نعمله حيث سنستخدم الدالة mergeAll، ولنرى النتيجة:

ملف app.component.ts

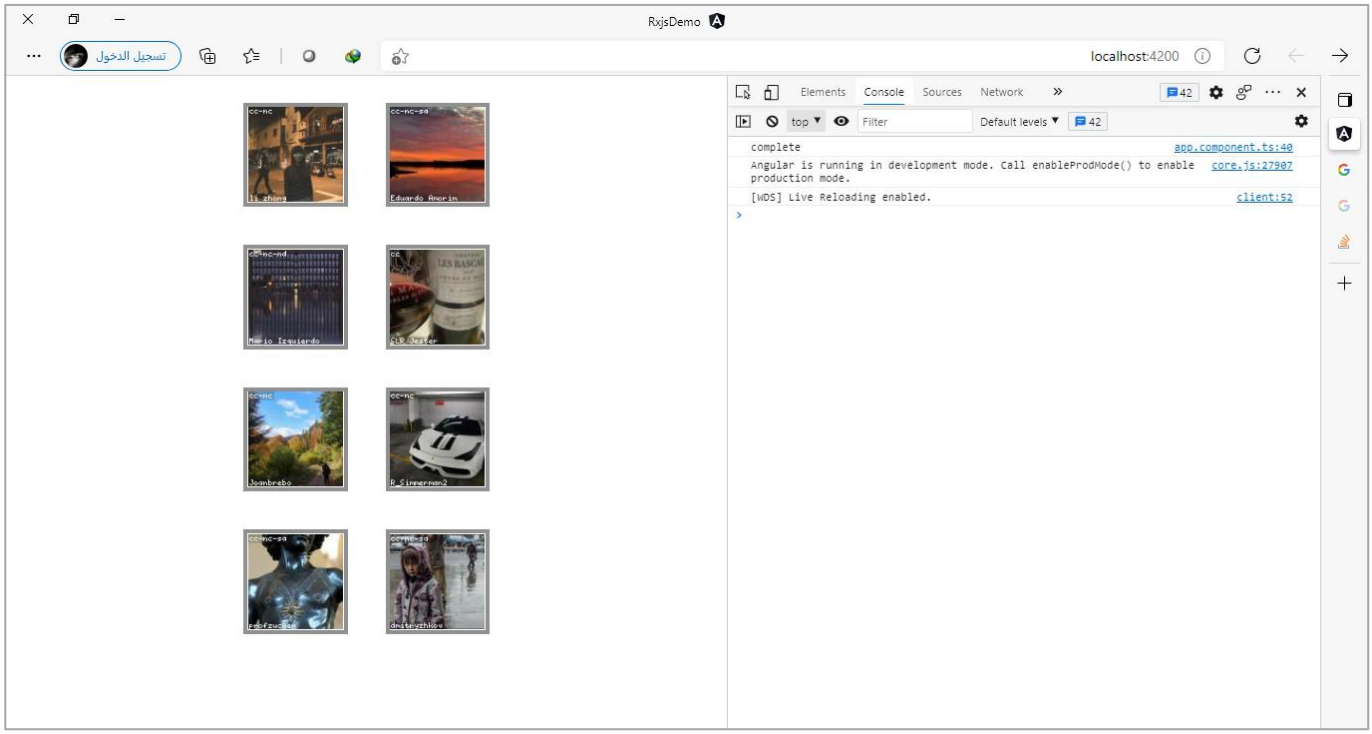
```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { map, mergeAll } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  public images: string[] = [];

  private categories = of('all','brazil','rio','paris','tree','saudi','hero','man');
  private imgUrl = of('https://loremflickr.com/100/100');

  ngOnInit(): void {
    this.categories
      .pipe(
        map((category) => {
          return this.imgUrl.pipe(
            map((url) => {
              return url + '/' + category;
            })
          );
        }),
        mergeAll()
      )
      .subscribe({
        next: (url) => this.images.push(url),
        error: (error) => console.log(error),
        complete: () => console.log('complete'),
      });
  }

}
```

نلاحظ أضفنا mergeAll لكي نقرأ Observable الداخل مع عمل Subscribe/Unsubscribe بشكل تلقائي، وسوف تلاحظ ايضاً ان عملها مشابهه لدالة merge حيث ستقوم بقراءة وعرض اول قيمة (في حالتنا هذه صورة) تأتي اليها وهكذا إلى ان يكتمل الـObservable.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



سوف تلاحظ عزيزي المتعلم أن عرض الصور لم يتم بشكل متسلسل وانما بشكل عشوائي اول قيمة تأتي هي التي يتم عرضها، إلى ان تكتمل جميع القيم.



اما من ناحية الدالة mergeMap فأمرها سهل فإذا فهمت mergeAll وmap فسوف تستنج عزيزي المتعلم ان هذه الدالة ماهي إلى دمج بين هاتين الدالتين، فلو استعرضنا الشفرة البرمجية السابقة، فسوف تلاحظ اننا استخدمنا الدالتين السابقتين في Outer Observable لذلك نستطيع الدمج بينهما واستخدام الدالة mergeMap عوضاً هاتين الدالتين، بحيث بدلاً من استخدام الاسطر البرمجية التالية:

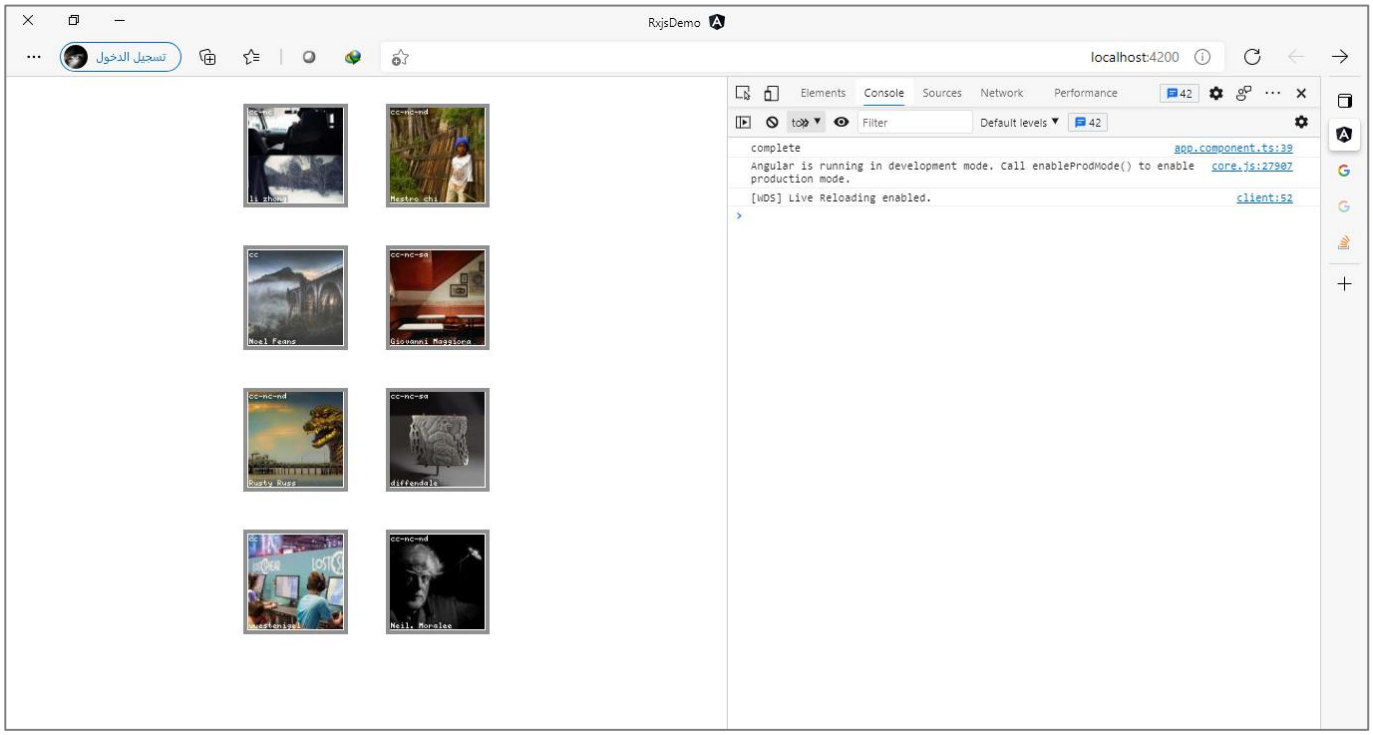
نقوم بتغييرها لتصبح بهذا الشكل، والنتيجة هي نفسها:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { map, mergeMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  public images: string[] = [];

  private categories = of('all','brazil','rio','paris','tree','saudi','hero','man');
  private imgUrl = of('https://loremflickr.com/100/100');

  ngOnInit(): void {
    this.categories
      .pipe(
        mergeMap((category) => {
          return this.imgUrl.pipe(
            map((url) => {
              return url + '/' + category;
            })
          );
        })
      )
      .subscribe({
        next: (url) => this.images.push(url),
        error: (error) => console.log(error),
        complete: () => console.log('complete'),
```

```
      });
  }
}
```

والنتيجة هي نفسها:



3-1-2-2- concatAll() / concatMap():

اعلم عزيزي المتعلم انك إذا فهمت الدالة concat التي تكلمنا عنها سابقاً والدالتين mergeAll وmap، فإنك سوف تستوعب وتفهم هذه الدالة بشكل سهل وسلس.

حيث هذه الدالة تتشابه مع الدالة concat بإنها تقوم بدمج اثنين Observable مع الاهتمام بالترتيب حيث تقرأ اول قيمة ومن ثم تنتظر إلى ان تنتهي ومن ثم تنتقل إلى القيمة التي تليها وهكذا إلى ان تنتهي من جميع القيم.

وبنفس الوقت هذه الدالة تعتبر من دوال Higher Order Mapping حيث تقرأ الInner Observable مع عمل له Subscriber/Unsubscribe بشكل تلقائي.

ولتوضيح لنقوم بتطبيق نفس المثال السابق ولكن مع بعض التعديلات البسيطة حيث سوف استخدم الدالة pipe ذات الاسم delay والتي هي الأخرى سنتكلم عنها لاحقاً ولكن هنا سوف امر عليها سريعاً، حيث استخدمتها لأقوم بتأخير قراءة كل قيمة لفترة زمنية محددة لكي نشاهد كيف تتعامل هذه الدالة مع القيم، وبنفس الوقت ولاختصار الوقت سوف استخدم الدالة concatMap مباشرة بدون الفصل بين الدالتين concatAll وmap، لكيلا نعيد ونكرر ما قلناه سابقاً.

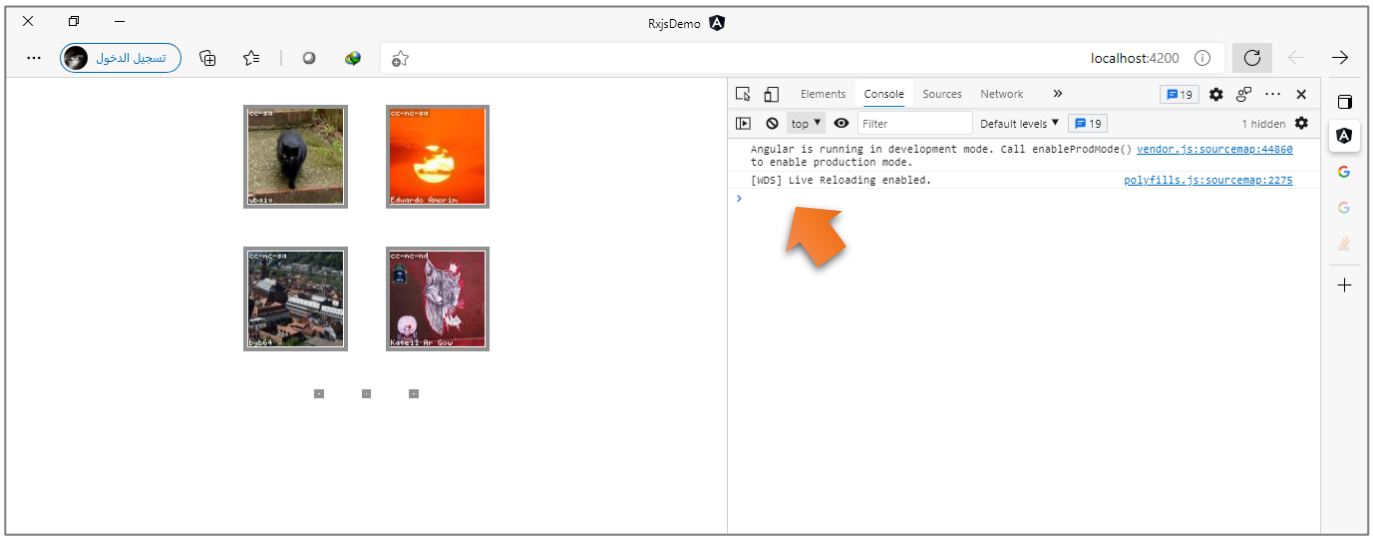
ملاحظة: ملفي الTemplate وملف Style لم أقم بأي تعديل عليهما والتعديل تم على ملف class فقط.

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable, of } from 'rxjs';
```

```
import { concatMap, delay, map } from 'rxjs/operators';

export interface Categories {
  name: string;
  delay: number;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  public images: string[] = [];

  private categories: Observable<Categories> = of(
    { name: 'all', delay: 300 },
    { name: 'brazil', delay: 500 },
    { name: 'rio', delay: 100 },
    { name: 'paris', delay: 0 },
    { name: 'tree', delay: 1000 },
    { name: 'saudi', delay: 700 },
    { name: 'hero', delay: 200 },
    { name: 'man', delay: 900 }
  );

  private imgUrl = of('https://loremflickr.com/100/100');

  ngOnInit(): void {
    this.categories
      .pipe(
        concatMap((category: Categories) => {
          return this.imgUrl.pipe(
            delay(category.delay),
            map((url) => {
              return url + '/' + category.name;
            })
          );
        })
      )
      .subscribe({
        next: (url) => this.images.push(url),
        error: (error) => console.log(error),
        complete: () => console.log('complete'),
      });
  }
}
```

والنتيجة في المتصفح سوف تلاحظ ان الصور تأتي بشكل متسلسل الصورة وراء الأخرى ولن يتم عمل complete في console إلا بعدما ان يتم الانتهاء من عرض جميع الصور، كالتالي:



switchAll() / switchMap -3-2-1-3:

ولعل هذه الدالة تعتبر من الدوال المهمة بعد الدالة map ولها استخداماتها المهمة وتُستخدم بكثرة من قبل المطورين، ولها عدة وظائف أولها انها تعتبر من دوال Higher Order Mapping وتؤدي نفس المهام التي تؤديها الدوال السابقة، اما وظيفتها الثانية وهي انها تقوم بعمل unsubscribe للObservable السابق ومن ثم تعمل subscribe او بصيغة أخرى تحويل switch للObservable الجديد. (كلام كبير ومعرفش حاجة عنه) لا تخف عزيزي المتعلم ولا تتوتر وأمسك اعصابك فالموضوع بسيط جداً ☺.

ولتوضيح لنفرض مثلاً انه لدينا قائمة تحتوي على مجموعة من الأزرار، زر لعرض بيانات الموظف وزر آخر لعرض بيانات الأقسام وزر ثالث لعرض بيانات العملاء وزر رابع لعرض بيانات المنتجات ...الخ، وكل زر من هذه الأزرار يقوم بجلب البيانات من قاعدة بيانات على السيرفر ولا يخفيك عزيزي المتعلم ان جلب هذه البيانات قد يستغرق بعض الوقت اعتماداً على سرعة الشبكة وأداء جهاز الحاسب والسيرفر ...الخ، لذلك قد يقوم مستخدم تطبيقك بالضغط على زر بيانات المنتجات وقبل اكتمال جلب البيانات قد يقوم بالضغط على زر بيانات العملاء، ففي هذه الحالة سوف يأتي الرد من السيرفر بعد فترة معينة ويعرض بيانات المنتجات وايضاً سيأتي الرد جالباً بيانات العملاء مما قد يؤدي إلى أخطاء في التطبيق لديك إذا لم تقم بعمل معالجة لهذه الطلبات جميعها، ناهيك عن استهلاك موارد الجهاز لدى العميل من خلال طلبات لا نحتاجها.

إذن ما هو الحل؟ هنالك عدة حلول منها وأسهلها استخدام مكتبة Rxjs والدالة الجميلة switchMap حيث تقوم في حال تم عمل emit لObservable جديد والObservable السابق لم يكتمل عندها ستقوم بعمل unsubscribe للObservable السابق وبنفس الوقت تعمل تحويل او switch او subscribe للObservable الجديد.

والعلاقة بين switchAll وswitchMap هي نفسها العلاقة بين الدوال السابقة، لذلك وابتعادً عن التكرار سوف أقوم بتطبيق المثال التالي باستخدام الدالة switchMap.

اما من ناحية المثال فهو تقريباً نفس فكرة المثال السابق، ولكن هنا سوف نضيف قائمة بشكل ديناميكي بحيث عناصر القائمة هي عبارة عن التصنيفات Categories وعندما يتم الضغط على أحد هذه التصنيفات سيتم الاتصال بالسيرفر عن طريق الرابط ومن ثم جلب صورة معينة بعد فترة زمنية معينة.

لذلك لنقوم بإضافة Markup اللازم في ملف Template، كالتالي:

ملف app.component.html

```
<div>
  <ul class="list-group list-group-flush">
    <li
      *ngFor="let category of categories | async"
      class="list-group-item"
      (click)="showImage(category)"
    >
      {{ category }}
    </li>
  </ul>
  <img [src]="imgUrl" *ngIf="imgUrl" />
</div>
```

نلاحظ وجود الدالة showImage (لك حرية اختيار الاسم الذي تُريده) ومررنا لها اسم التصنيف بحيث إذا قام المستخدم بالضغط على عنصر معين من القائمة يتم ارسال اسم هذا العنصر كنص string إلى هذه الدالة.

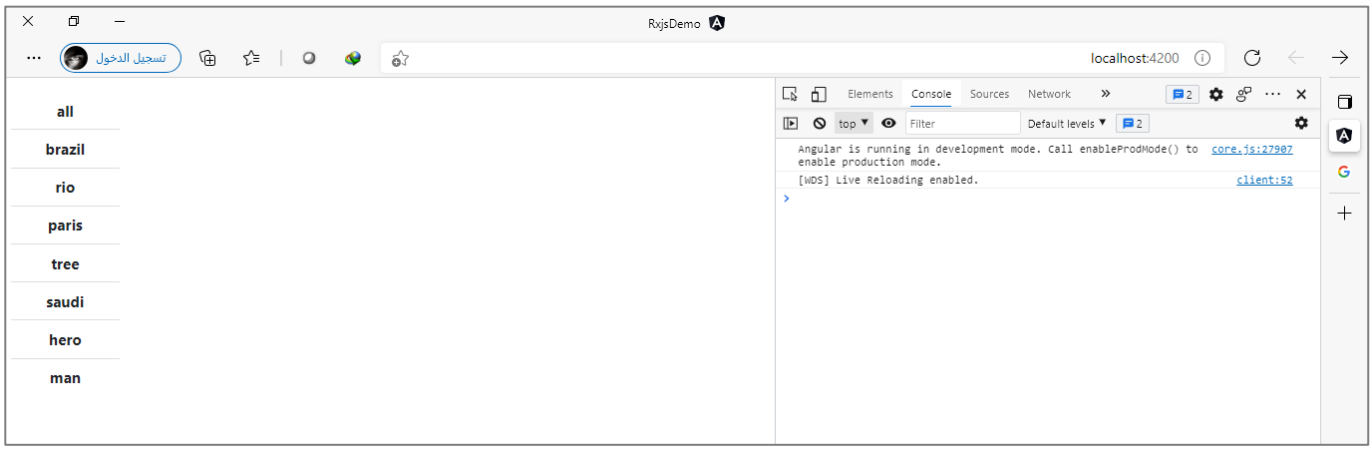
ولنضيف الآن بعض التنسيقات في ملف Style، كالتالي:

ملف app.component.scss

```
div {
  box-sizing: border-box;
  display: flex;
  flex-wrap: nowrap;
  flex-direction: row;
  justify-content: flex-start;
  align-items: flex-start;
  align-content: stretch;
}
ul {
  margin-top: 15px;
  width: 120px;
}
li {
  text-align: center;
  cursor: pointer;
  margin-left: 5px;
}
```

```
img {
  border: 5px solid #919191;
  margin: 0 auto;
  width: 200;
  height: 300px;
  margin-top: 30px;
}
```

ولنشاهد التطبيق إلى الآن، كالتالي:



تما الآن لنكتب محتوى الدالة showImage، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { delay, switchMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  public imgUrl = '';
  public categories = of(['all','brazil','rio','paris','tree','saudi','hero','man']);

  showImage(category: string): void {
    this.categories
      .pipe(                                          Outer Observable
        switchMap(() => {
          return of('https://loremflickr.com/100/100/' + category).pipe(
            delay(1000)
                                                  Inner Observable
          );
        })
      )
      .subscribe({
        next: (url) => (this.imgUrl = url),
```

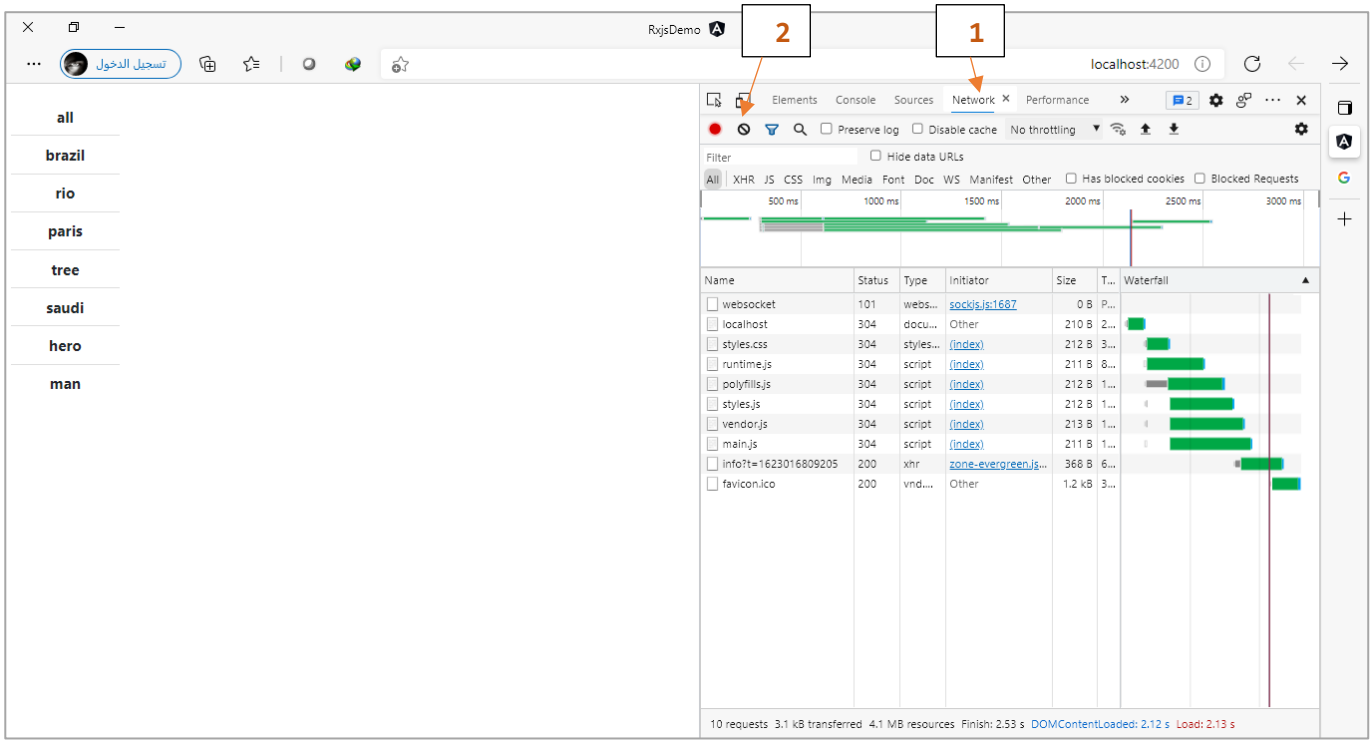
```
      });
  }
  ngOnInit(): void {}
}
```

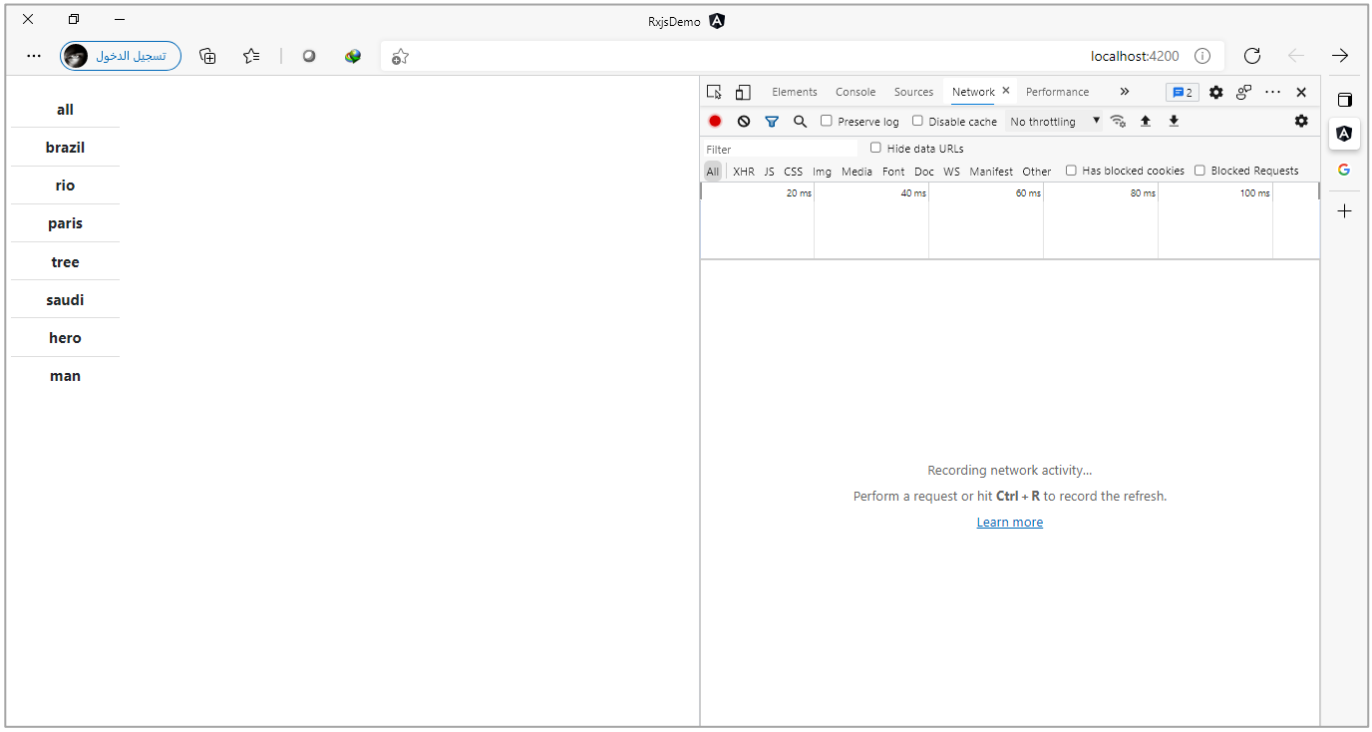
نلاحظ ان Outer Observable يستقبل التصنيفات وعن طريق الدالة switchMap نقوم بقراءة وجلب الصورة المحددة، ونلاحظ ان switchMap لم امرر لها أي قيمة والسبب ان القيمة (اسم التصنيف) تم تمريره عن طريق البارامتر category في الدالة showImage، لذلك نكتفي بهذا البارامتر حيث نقوم بتمريره إلى الرابط لكي يتم جلب الصورة.

وبذلك عندما يتم الضغط على عنصر في القائمة وليكن العنصر tree، فإن الدالة showImage سيتم تنفيذها وتمرير الاسم tree لها، وهذه الدالة ستقوم بتشغيل Outer Observable وتنفذ الدالة switchMap حيث ستقوم بتنفيذ Inner Observable وارسال الرابط متضمناً الصنف tree، وقمنا بمحاكاة ان جلب هذه الصورة قد يأخذ بعض الوقت عن طريق الدالة delay.

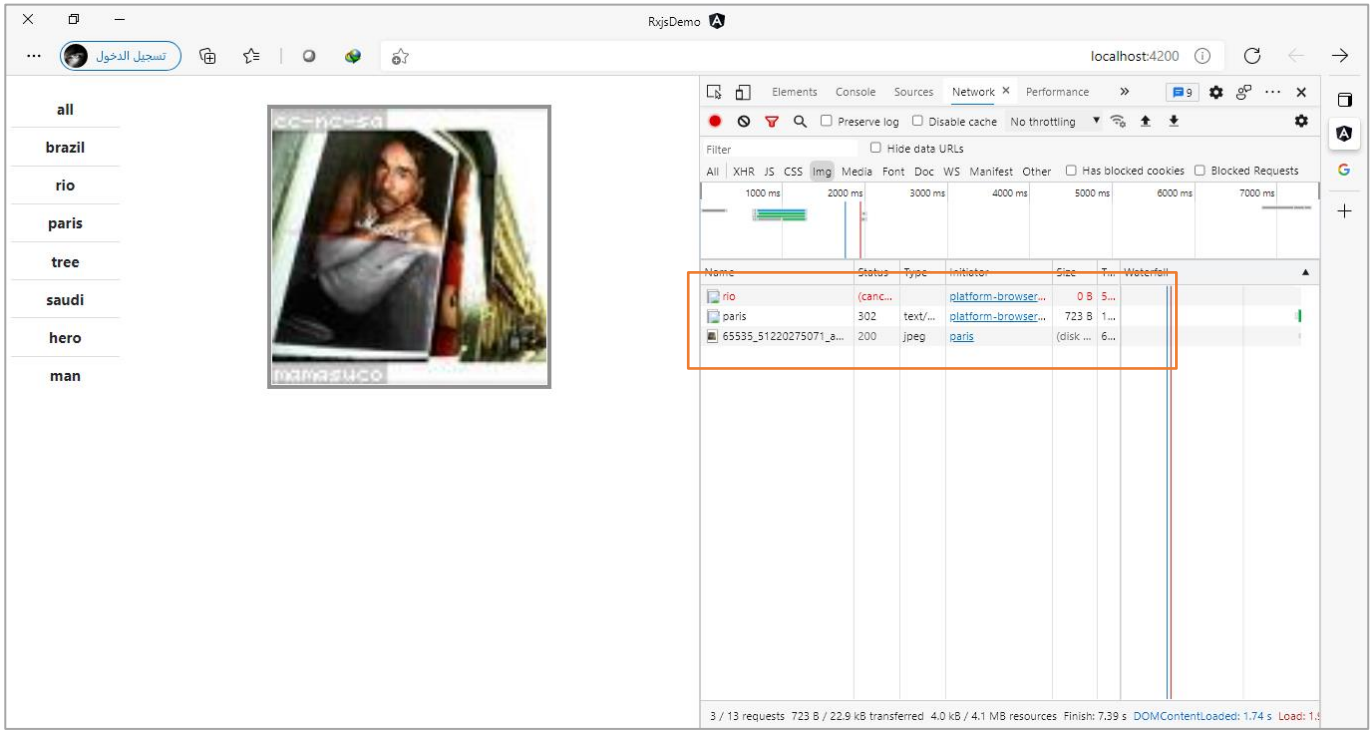
وهكذا مع كل ضغطة من المستخدم يتم تنفيذ جميع ما تم ذكره سابقاً، وعندما يتم الضغط على عنصر معين وبشكل سريع يتم الضغط على عنصر آخر فإن الدالة switchMap سوف تقوم بإلغاء الObservable الأول وتقوم بعمل Subscribe للObservable الجديد.

ولكي نتأكد من ان ما عملناه يعمل بشكل صحيح، لنقوم بتشغيل المتصفح، ومن ثم فتح أدوات المطور ونختار التبويب Network ومن ثم نعمل clear للملفات المحملة لكي نستطيع ان نشاهد ما نُريده بشكل واضح، كما في الصورة التالية:





ولنقوم الآن بالضغط على عنصر rio من القائمة وبنفس الوقت وبشكل سريع لنضغط على عنصر Paris، ولنرى النتيجة:



نلاحظ ان ملف rio تم الغاء request من السيرفر وتم الاتصال بالرابط الآخر وجلب صورة تحت التصنيف Parise.

exhaustAll() / exhaustMap() -4-2-1-3:

وهذا النوع من الدوال المهمة ولها استخداماتها، فهي بالإضافة إلى انها من دوال Higher Order Mapping تقوم بعكس المهمة التي تقوم بها switchMap، حيث تقوم بإخذ أول Observable ومن ثم تقوم بتجاهل أي Observable يحدث اثناء

وجود الObservable الأول وعندما تنتهي منه ويتم عمل complete له تستقبل أي Observable آخر في حال وجوده، وهكذا إلى ان تنتهي من جميع الObservables.

وأرجو الانتباه عزيزي المتعلم إلى انها لا تجعل Observable الذي يأتيها وهي تقوم بمعالجة الObservable الحالي ينتظر وانما تقوم بتجاهله – الغاءه – فهي بما انها تقوم بمعالجة الObservable الحالي وهذا الObservable لم ينتهي ستقوم بإلغاء أي طلب او Observable جديد يصلها. وهذا هو لب وأهمية هذه الدالة.

اما من ناحية استخداماتها، فهي متعددة منها على سبيل المثال لا حصر، عندما تذهب لشراء أغراض من السوبرماركت وأردت ان تدفع قيمة مشترياتك بالبطاقة البنكية التي تعتمد على wifi، فإن جهاز الدفع عندما تضع بطاقتك عليه يقوم باستقبال اول طلب ولو حاولت ان تُعيد وضع البطاقة مرات متعددة فإن التطبيق الموجود على جهاز المشتريات سوف يتجاهله تماماً بما انه يقوم بمعالجة طلب حالياً، وعندما ينتهي هذا الطلب بالرفض او القبول بمعنى انتهى او تم عمل له complete عندها يستقبل طلب جديد.

قد تستغرب من هذا الاستخدام وحقيقة انا لا اعلم هل التطبيق الذي تم برمجته على هذا النوع من الأجهزة للقيام بمثل هذه المهام يستخدم مكتبات Reactive Programming ام لا.

ولكن الذي اعلمه جيداً ان تقنيات Reactive Programming لا تقتصر على الجافا سكربت وانما يتم استخدامها من قبل لغات برمجة لها ثقلها ووزنها بالسوق مثل الجافا والبايثون و++c و#c وغيرها من اللغات المشهورة، بنفس المفاهيم والدوال وبنفس اسمائها، لذلك لا تستغرب عزيزي المتعلم ان تم استخدام هذه الدالة لتعامل مع هذا النوع من الطلبات.

ومن الاستخدامات الأخرى، طلبات Server سواء بالإضافة او الحذف او التعديل او القراءة، والتي تُسمى CRUD، فمثلاً عندما يقوم المستخدم بتعبئة بيانات معينة ويتم الضغط على زر الحفظ لإضافة هذه البيانات في قاعدة البيانات، فمن الوارد ان يقوم المستخدم بالضغط مرات متعددة على زر الحفظ، مما يجعل هنالك طلبات زائدة على السيرفر ولك ان تتخيل عزيزي المتعلم لو ان تطبيقك يستخدمه عشرات او مئات الألوف بنفس الوقت، فكيف ان هذا الامر قد يؤثر على أداء تطبيق ويستهلك موارد السيرفر.

وهذا الامر له حلول متعددة منها ان يتم تعطيل زر الحفظ عند اول ضغطة ومن ثم عند الانتهاء من الحفظ يتم تفعيل الزر مرة أخرى، وهذا حل جيد ولكن ثق عزيزي المتعلم انه وفي حال قمت بكتابة الامر البرمجي الذي يقوم بفعل هذا الأمر إلا انه هنالك حالات من المستخدمين لديهم أجهزة ذات أداء ضعيف نسبياً او هنالك بعض الإشكاليات التي تؤثر على أداء اجهزتهم، وهذا الضعف ينتج عنه تأخير لفترة زمنية معينة قبل تعطيل هذا الزر وهذا التأخير قد يجعل المستخدم يقوم بالضغط أكثر من مرة.

لذلك هنالك دوال جاهزة تقدمها لنا تقنيات Reactive Programming والمتمثلة في حالتنا هذه بمكتبة Rxjs وهي دالتي exhaustMap او exhaustAll (العلاقة بينهما مشابهة للعلاقة بين الدوال الأخرى)، وهاتين الدالتين تستقبلان اول طلب

او بصيغة أدق اول Observable وتتجاهل أي Observable آخر يتم أرساله إلى ان يتم اكتمال هذا الطلب، ومن ثم تستقبل أي Observable آخر سيتم أرساله بعد اكتمال الطلب (وليس تم ارساله والطلب الأول لازال موجود).

ولاحترافية أكثر تستطيع عزيزي المتعلم الدمج بين هاتين الطريقتين أي تستخدم إحدى الدالتين exhaustMap او exhaustAll وبنفس الوقت تقوم بتعطيل الزر إلى ان يكتمل الطلب.

اما الآن لنعطي مثال لتوضيح ما قيل سابقاً، سوف نقوم بقراءة API باستخدام دالة ajax وهي إحدى الدوال التي تقدمها لنا مكتبة Rxjs، لقراءة API واجراء جميع عمليات CRUD عليه.

ولكن لم أتطرق إليه في هذا الكتاب لأننا في عالم الAngular لدينا ما يسمى HttpClient والتي تُغنينا عن هذه الدالة، وسوف اخصص لها فصل كامل من هذا الكتاب اشرح هذه التقنية واهميتها مع إعطاء مقدمة ومدخل إلى API بإذن الله.

ولكن وبما اننا لم نصل إلى تقنيات HttpClient فسوف استخدم دالة ajax ولا تقلق ولا تحاول فهمها فقط اعلم انها تُساعدنا في جلب بيانات معينة على سيرفر ما في هذا العالم عن طريق تمرير رابط له.

اما الرابط فهو مقدم من موقع jsonplaceholder، حيث يُقدم هذا الموقع مجموعة من البيانات على شكل روابط Api للاختبار والتجربة فقط.

وقبل الانتقال إلى الشفرة البرمجية يجب ان أشير إلى انني اضفت الدالة delay ومررت لها القيمة 2000 لكي أقوم بتأخير الشفرة البرمجية ثانيتين، لمحاكاة التأخير الذي قد يحدث في السيرفر قبل اكتمال الطلب، والدالة mergeMap لكي نتعامل مع Higher Order Mapping.

ولننتقل الآن إلى الشفرة البرمجية، كالتالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { delay, map, mergeMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('btnGetPosts', { static: true }) btnGetPosts: ElementRef;
  ngOnInit(): void {
    fromEvent(this.btnGetPosts.nativeElement, 'click')
      .pipe(
        mergeMap(() => {
          return ajax
            .get<any>('https://jsonplaceholder.typicode.com/posts/')
```

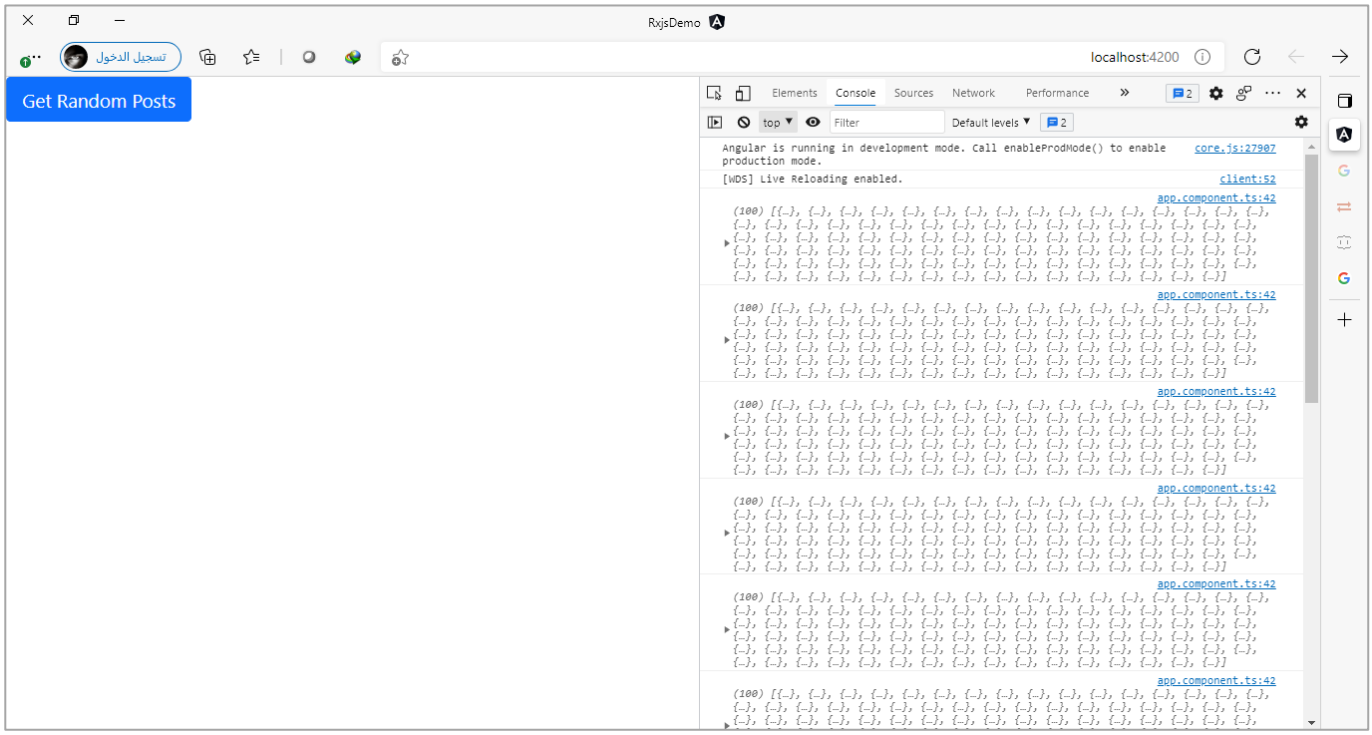
```
            .pipe(
              map((posts: any) => posts.response),
              delay(2000)
            );
        })
      )
      .subscribe((data) => {
        console.log(data);
      });
  }
}
```

وفي ملف Template نضيف الشفرة البرمجية التالية:

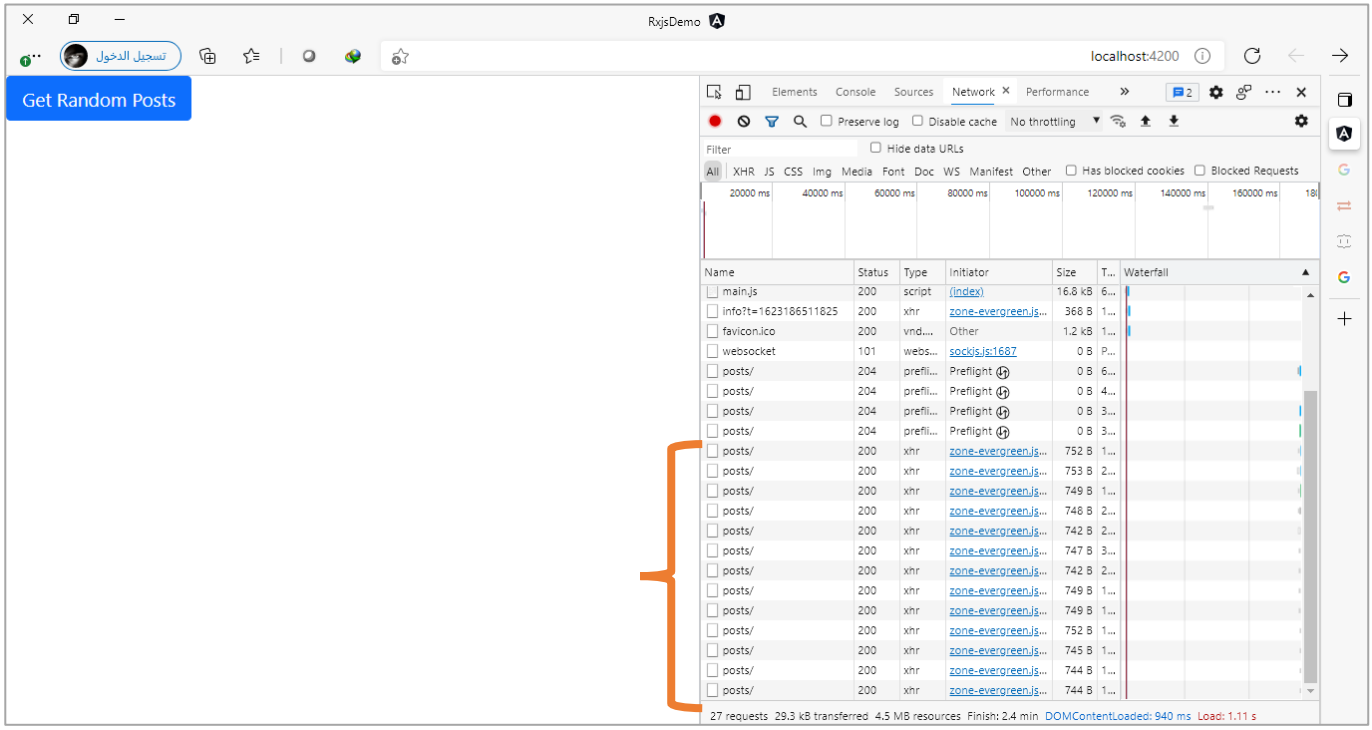
ملف app.component.html

```
<button #btnGetPosts class="btn btn-primary btn-lg">
  Get Random Posts
</button>
```

اما الآن لنشاهد النتيجة في المتصفح:



تلاحظ عزيزي المتعلم كمية الطلبات إلى السيرفر مع كل ضغطة على زر جلب البيانات، ولنذهب إلى التبويب network لكي نرى الطلبات بشكل أوضح:



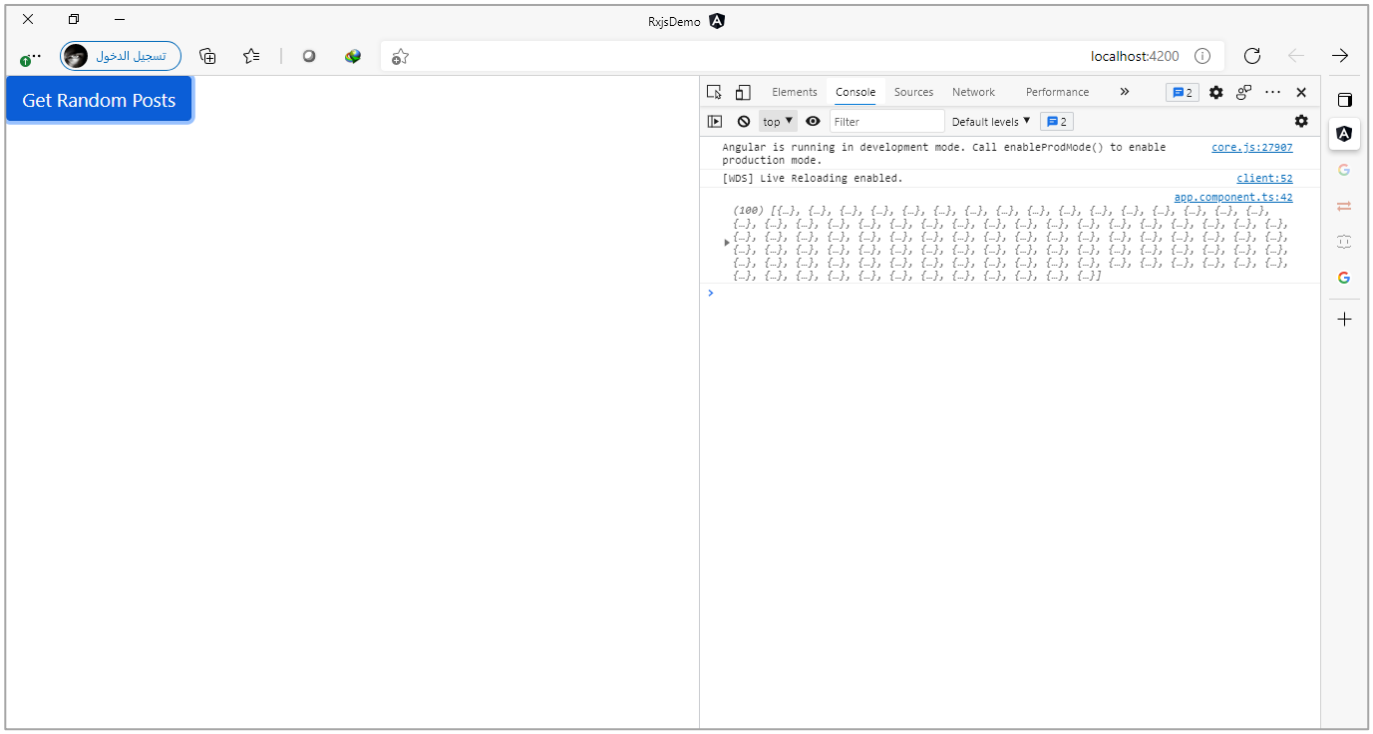
لاحظ ايضاً عزيزي المتعلم عدد الطلبات، وذلك يدل على ان دالة mergeMap صحيح انها تتعامل مع الObservables المتداخلة او ما يسمى Higher Order Observable، ولكن لا تصلح لهذا النوع من Observable لأن لكل دالة استخداماتها التي يجب عليك عزيزي المتعلم ان تعرف متى وكيف تستخدم الدالة التي تُلبي احتياجك.

لذلك لنستبدل هذه الدالة بالدالة exhaustMap، ولنرى النتيجة، كالتالي:

ملف app.component.ts

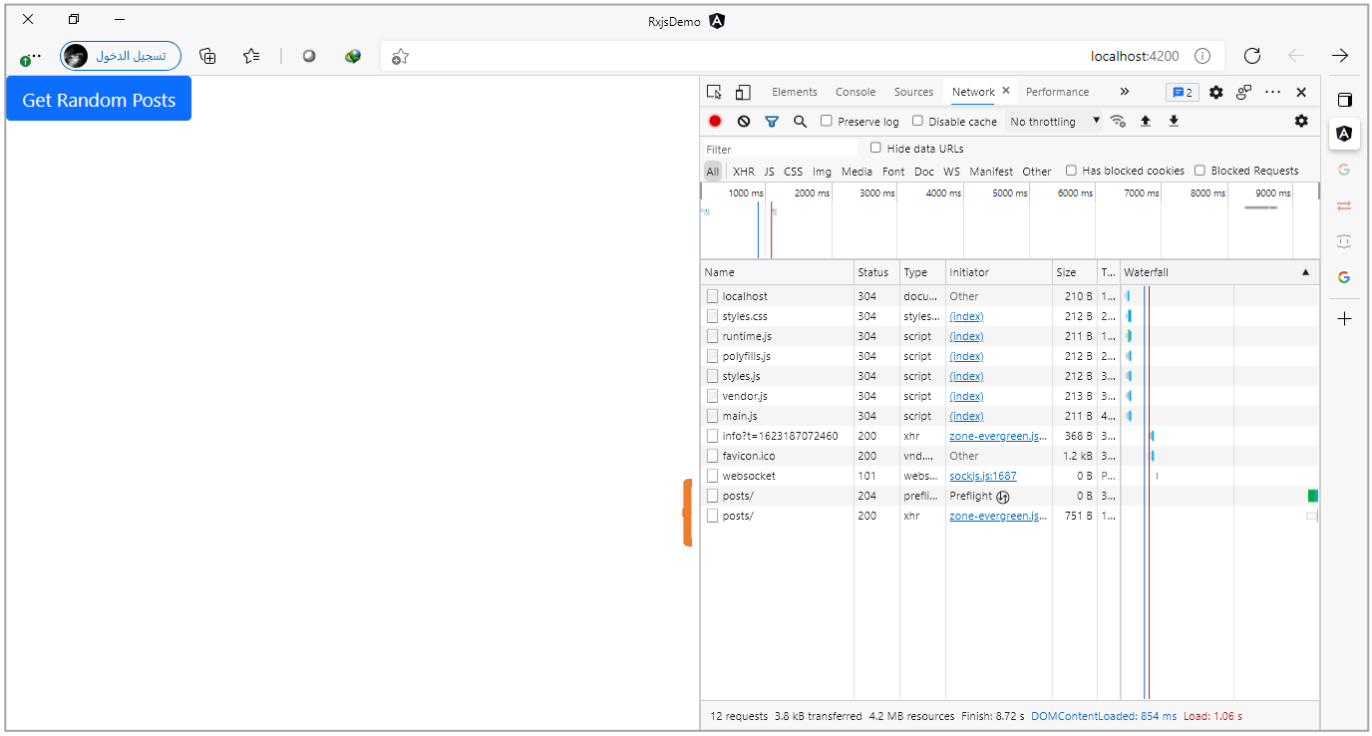
```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { delay, map, exhaustMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('btnGetPosts', { static: true }) btnGetPosts: ElementRef;
  ngOnInit(): void {
    fromEvent(this.btnGetPosts.nativeElement, 'click')
      .pipe(
        exhaustMap(() => {
          return ajax
            .get<any>('https://jsonplaceholder.typicode.com/posts/')
            .pipe(
              map((posts: any) => posts.response),
              delay(2000)
            );
```

```
        })
      )
      .subscribe((data) => {
        console.log(data);
      });
  }
}
```



نلاحظ عزيزي المتعلم انه على الرغم من الضغط على الزر أكثر من مرة إلا انه تم تجاهلها جميعاً طول فترة الثانيتين، وبعد انتهاء هذا الوقت واكتمال الObservable فإن أي ضغطة سوف تُعيد نفس الخطوات معها وهكذا.

اما الآن لنذهب إلى تبويب Network، ولنشاهد الطلبات والرد من السيرفر، كالتالي:



كما تلاحظ عزيزي المتعلم ان جميع الطلبات تم تجاهلها من قبل هذه الدالة ولم تُرسل من الأساس إلى السيرفر كأنها لم تحدث اصلاً، وذلك بعكس الدالة switchMap التي تقرأ الطلبات ولكن عند حدوث طلب جديد تقوم بإلغاء الطلب والتحويل إلى الطلب الجديد او بمعنى اصح إلى الObservable الجديد.

وايضاً نستطيع الغاء تفعيل الزر وعند اكتمال الObservable نُعيد تفعيله مرة أخرى، كالتالي:

app.component.ts ملف

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { delay, map, exhaustMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('btnGetPosts', { static: true }) btnGetPosts: ElementRef;
  disabled = false;

  ngOnInit(): void {
    fromEvent(this.btnGetPosts.nativeElement, 'click')
      .pipe(
        exhaustMap(() => {
          this.disabled = true;
          return ajax
            .get<any>('https://jsonplaceholder.typicode.com/posts/')
            .pipe(
```

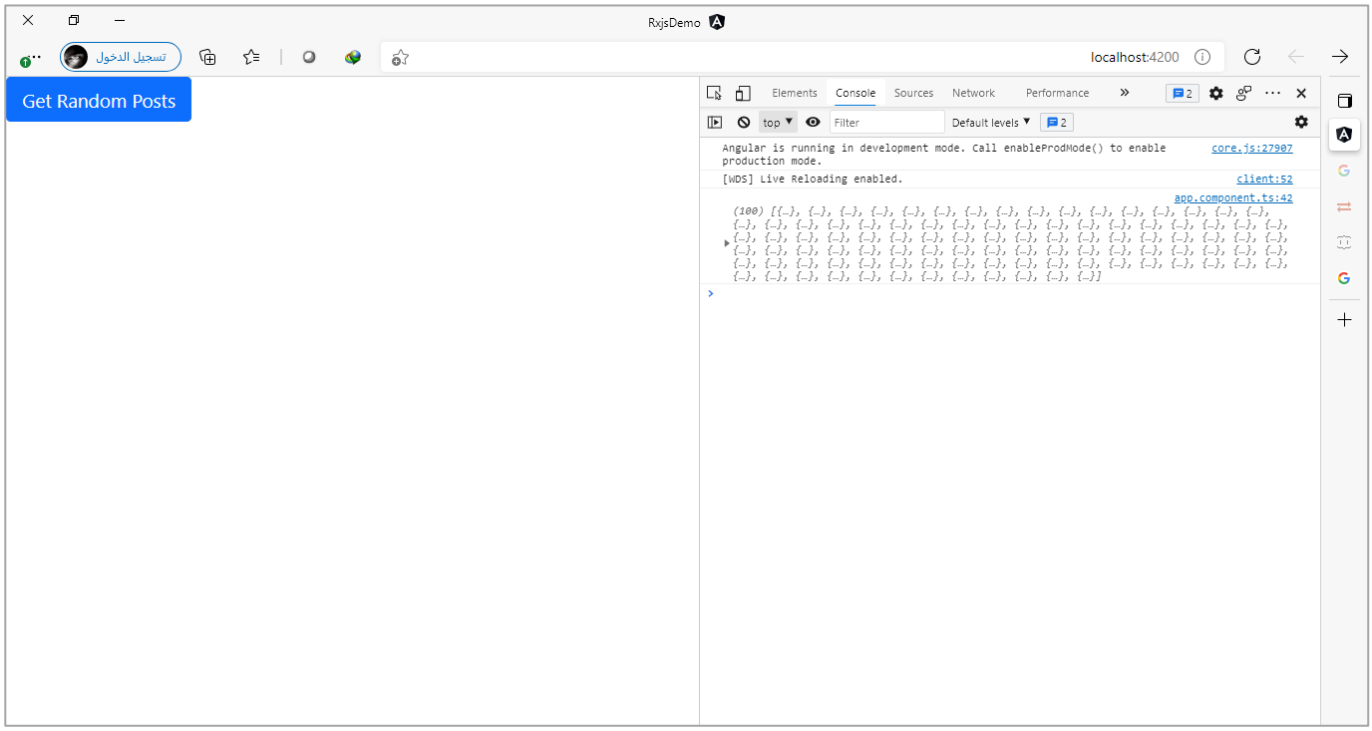
```
            map((posts: any) => posts.response),
            delay(2000)
          );
      })
    )
    .subscribe((data) => {
      console.log(data);
      this.disabled = false;
    });
  }
}
```

وفي ملف Template نضيف الMarkup التالي:

ملف app.component.html

```
<button #btnGetPosts class="btn btn-primary btn-lg" [disabled]="disabled">
  Get Random Posts
</button>
```

اما النتيجة في المتصفح، فتكون كالتالي:



سوف تلاحظ انه عند الضغط على هذا الزر تم الغاء تفعيله وعند اكتمال الObservable تم إعادة تفعيله مرة أخرى.

وفي حال أردت استخدام exhaustAll، فتستطيع ذلك بكل بساطة، كالتالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { delay, map, exhaustMap, exhaustAll } from 'rxjs/operators';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('btnGetPosts', { static: true }) btnGetPosts: ElementRef;
  disabled = false;
  ngOnInit(): void {
    fromEvent(this.btnGetPosts.nativeElement, 'click')
      .pipe(
        map(() => {
          this.disabled = true;
          return ajax
            .get<any>('https://jsonplaceholder.typicode.com/posts/')
            .pipe(
              map((posts: any) => posts.response),
              delay(2000)
            );
        }),
        exhaustAll()
      )
      .subscribe((data) => {
        console.log(data);
        this.disabled = false;
      });
  }
}
```

ولو قمت بتشغيل التطبيق فسوف تشاهد نفس النتيجة السابقة.

وبذلك نكون أنهينا اهم دوال مكتبة Rxjs وأكثرها استخداماً، وفي الجزء التالي سنتطرق إلى دوال تأتي بالمرتبة الثانية من ناحية الأهمية وشيوع الاستخدام، وهي دوال Scanning Operators.

Scanning Operators -3-1-3:

المقصود بها الدوال التالية:

- scan()
- mergeScan()
- switchScan()

وهذه الدوال تأتي في الأهمية بعد دوال Mapping بجميع اشكالها، وتكون اقل منها شيوعاً واستخداماً، ولكن قد نحتاجها وخصوصاً دالة scan.

للوهلة الأولى عندما تقرأ عن هذه الدالة سوف تجد انها دالة رياضية بالدرجة الأكبر حيث تقوم بعمل جمع لرقمين وكُل عملية جمع تعمل emit لناتج جمع هذه القيم، وهي مشابهه لذلك نوعاً ما لدالة reduce في مصفوفات الجافا سكريبت.

ولكن في الحقيقة عملها هنا يتعدى موضوع اجراء بعض العمليات الحسابية الرقمية، وانما هي تتعامل مع اغلب أنواع البيانات بحيث نستطيع اجراء عملية جمع لمجموعة من البيانات سواء على شكل كائن او مصفوفة، فعلى سبيل المثال لو كان لدينا Observable يحتوي على بيانات مجموعة من الموظفين، وتم استقبالها في Subscriber ومن ثم عرضها في التطبيق، ففي حالة تم عمل emit لبيانات أخرى بنفس هذا الObservable فإن هذه الدالة تقوم بدمج البيانات الجديدة مع البيانات القديمة، وهكذا كلما وجد بيانات جديدة يتم دمجها.

ولعل هذه الحالة هي الحالة الشائعة لاستخدامات هذه الدالة.

اما الآن لنستعرض هذه الدوال الثلاثة مع الأمثلة والشرح المُفصل بالتحديد لدالة scan.

3-1-3-1- scan():

هذه الدالة أكثر دوال scanning شيوعاً من ناحية الاستخدام، وكما أشرت سابقاً نستخدمها في الغالب لإدارة ودمج البيانات لحظياً بحيث أي بيانات جديدة يتم دمجها مع السابقة لها ومن ثم نعمل emit للبيانات الجديدة بعد الدمج، وهكذا كلما تُوجد بيانات جديدة فسيتم دمجها مع السابقة.

اما الآن لنعطي مثال لتوضيح، وهذا المثال سيكون مثال بسيط لشرح هذه الدالة بشكلها المبسط، وهو عبارة عن مجموعة من البيانات الرقيمة بحيث نستخدم هذه الدالة لجمع هذه البيانات.

وقبل استعراض المثال يجب التوضيح ان الدالة scan تستقبل بارامترين الأول وهو دالة بدون اسم function anonymous وهذه الدالة التي بدون اسم تستقبل هي الأخرى بارامترين الأول هو المجمع accumulator والثاني هو القيمة value، والمقصود بالمجمع هو الذي يقوم بجمع (دمج) القيم قبل ارسالها، والثاني value فتخزن فيه القيمة الجديدة القادمة من Observer، اما البارامتر الثاني لدالة scan فيتم فيه تخزين القيمة الابتدائية للمجمع accumulator.

ولتوضيح لنستعرض المثال التالي:

ملف app.component.ts

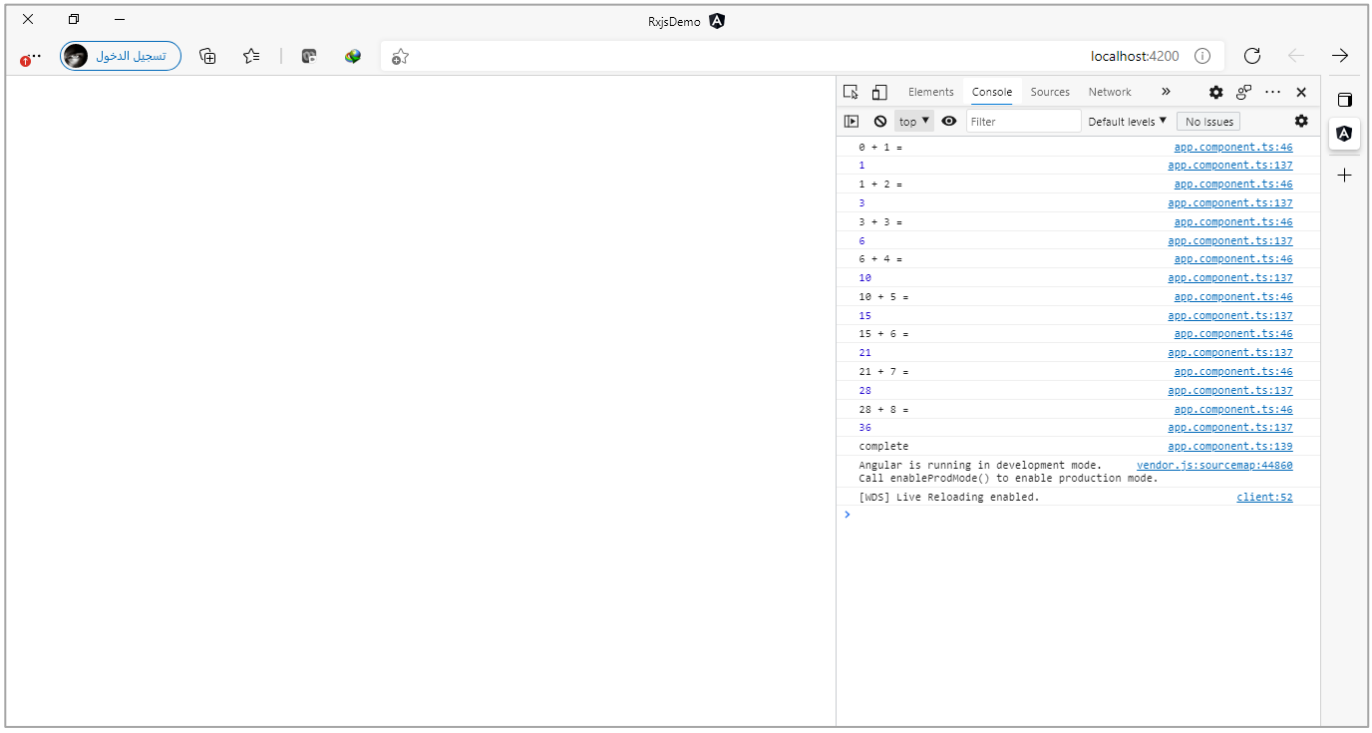
```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { scan } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
```

```
  ngOnInit(): void {
    of(1, 2, 3, 4, 5, 6, 7, 8)
      .pipe(
        scan((acc: number, value: number) => {
          console.log(`${acc} + ${value} = `);
          return acc + value;
        }, 0)
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم يوجد لدينا Observable قمنا بإنشائه عن طريق الدالة of ومن ثم قمنا بتمرير هذا Observable لدوال pipe وبالتحديد بالدالة scan حيث مررنا لها بارامترين الأول دالة وهذه الدالة ايضاً تستقبل بارامترين الأول يُمثل المُجمع واسميته acc - لك حرية اختيار الاسم الذي تُريده – اما الثاني فيمثل القيم الجديدة القادمة من الObservable واسميته value – لك حرية اختيار الاسم الذي تُريده – اما البارامتر الثاني لدالة scan مررنا له القيمة صفر أي ضع قيمة مبدئية للمجمع صفر والذي يمثله في مثالنا هذا البارامتر acc.

ولو تتبعنا سير البرنامج فسنجد ان اولاً سيقوم Observer بعمل emit للقيمة 1 وفي الدالة scan سيضع هذه القيمة في البارامتر value ومن ثم سيضع القيمة المبدئية للمُجمع acc بصفر، ومن ثم يجمع القيمة في كلا البارامترين والنتيجة يعمل لها emit في subscription لكي يقرأها subscriber ويتم عرضها في console، ومن ثم الObserver سيقوم بعمل emit للقيمة التالية في Observable وهي 2، وعندها سيضع هذه القيمة في البارامتر value ويجمع هذه القيمة مع القيمة الموجودة سابقاً في المجمع acc وهي 1 ومن ثم يعمل emit لنتيجة وهكذا إلى ان ينتهي من جميع القيم.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



ولننتقل الآن إلى المثال الثاني، وفيه نبين الاستخدام الحقيقي لهذه الدالة، حيث لنفترض انه لدينا نظام للإدارة رسائل الخطأ التي تظهر للمستخدم في تطبيقنا بحيث أي خطأ يحدث سيتم اكتشافه من قبل هذا النظام وارسال رسالة إلى المصفوفة معينة يتم تعريفها لعرضها للمستخدم، وايضاً في حال اكتشاف خطأ آخر يتم ارسال رسالة أخرى لهذه المصفوفة، بحيث يكون لدينا مصفوفة واحدة نجمع فيها جميع رسائل الخطأ.

بطبيعة الحال لن نتطرق لكيفية اكتشاف الأخطاء والتعامل معها، لأنني سوف اتطرق بالتفصيل لاحقاً في كتاب منفصل عن هذا الأمر والذي يدور محتواه من خلال مثال شامل لبناء نظام والذي من خلاله تستطيع عزيزي المتعلم ترسيخ اهم المفاهيم في هذه السلسة، لذلك سوف نقوم بتركيز على كيفية إدارة وعرض هذه الرسائل باستخدام الدالة scan، ولنفرض انه تم اكتشاف الأخطاء ومن ثم بعد زمن يتم عمل emit لخطأ آخر وأخيرا بعد فترة زمنية معينة يتم عمل emit لخطأ آخر، ونُريد ان يتم عمل دمج وإدارة لهذه الرسائل باستخدام الدالة scan، ولنقوم الآن باستعراض المثال، كالتالي:

ملف app.component.html

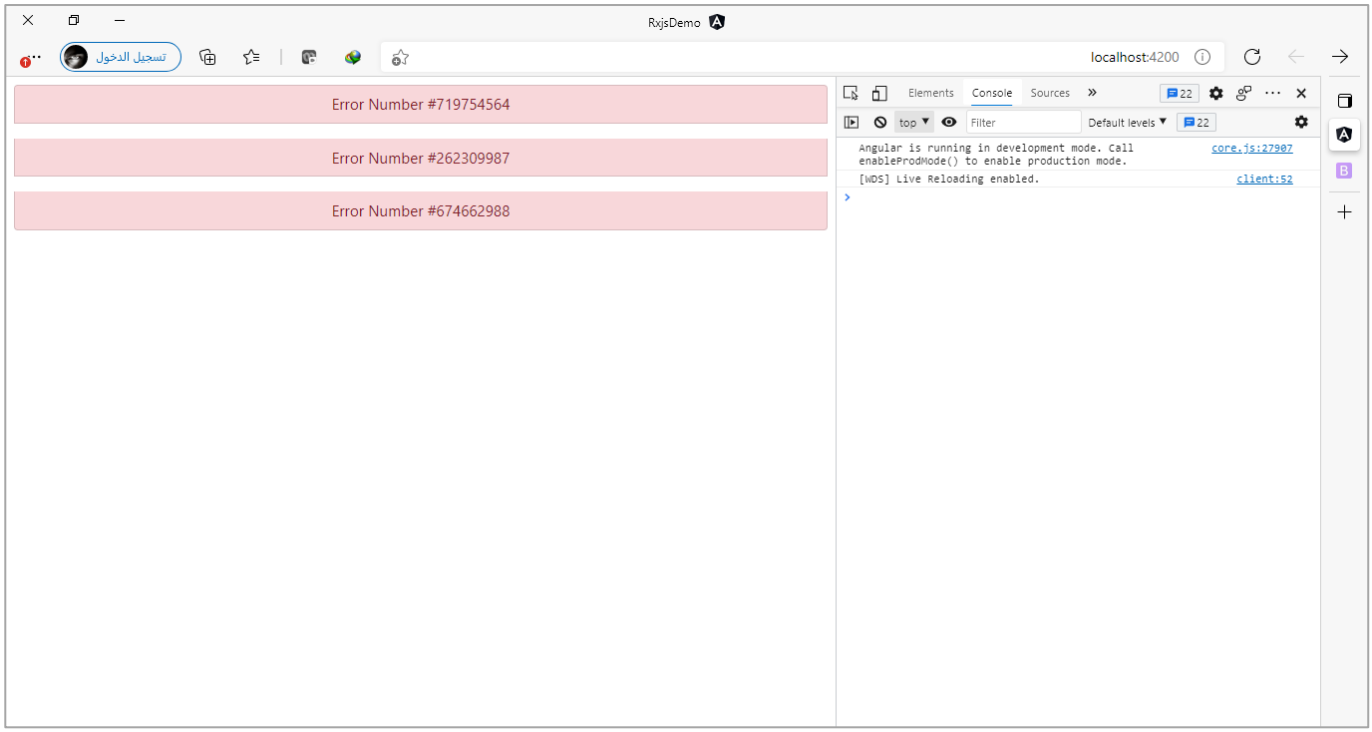
```
<ul class="list-group">
  <li
    class="list-group-item list-group-item-danger m-2 text-center"
    *ngFor="let message of messages"
  >
    Error Number #{{ message }}
  </li>
</ul>
```

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
import { scan } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  private msg = new Subject<number>();
  public messages = [];
  constructor() {}

  ngOnInit(): void {
    this.msg
      .pipe(
        scan((msgs: number[], newMsg) => {
          return [...msgs, newMsg];
        }, [])
      )
      .subscribe({ next: (value) => (this.messages = value) });
```

```
    setTimeout(() => {
      this.msg.next(Math.floor(Math.random() * 1000000000));
    }, 1000);
    setTimeout(() => {
      this.msg.next(Math.floor(Math.random() * 1000000000));
    }, 2000);
    setTimeout(() => {
      this.msg.next(Math.floor(Math.random() * 1000000000));
    }, 3000);
  }
}
```

كما تلاحظ عزيزي المتعلم قمنا باستخدام الدالة Subject لكي نقوم بعمل emit للقيم من خارج الObservable، وايضاً لكي نعمل لها subscribe، كما نلاحظ اننا قمنا بمحاكاة ان نظام مكتشف الأخطاء قد قام باكتشاف خطأ وارسال رسالة بعد اول ثانية ومن ثم بعد ثانيتين تم ارسال رسالة خطأ ثانية وبعد ثلاث ثواني ارسل رسالة الخطأ الثالثة، واخيراً قمنا بعمل دمج وإدارة لهذه الرسائل عن طريق الدالة scan، والنتيجة وضعناها في مصفوفة اسميتها messages، ولنشاهد النتيجة في المتصفح، كالتالي:



اما المثال التالي ومصدره على هذا الرابط RxJS Operator - Angular Counter with Scan, merge, startWith & tap - With Pause and Reset - YouTube، وفيه يتم ترسيخ لمفاهيم دالة scan بالإضافة لمراجعة لبعض الدوال والمفاهيم السابقة، وتقوم فكرة المثال بإنشاء عداد counter بالاعتماد على تقنيات مكتبة rxjs، حيث سنستخدم BehaviorSubject لكي نمرر قيمة مبدئية وايضاً لكي نستطيع ان نعمل emit للقيم من خارج الObservable وأخيراً لنتعامل معها كObservable ونستطيع ان نعمل لها Subscription بالإضافة إلى الاستفادة من دوال pipe، اما القيمة التي نمررها إلى هذا Observable فتكون عبارة عن كائن Object، يحتوي على خاصيتين الأولى من النوع boolean ونحدد هل العداد متوقف مؤقتاً او

مستمر، وايضاً القيمة الحالية، وبما اننا مررنا Object فسوف نقوم بعمل دمج القيم عن طريق الدالة scan ككائن وليس كمصفوفة كما هو الحال في المثال السابق، وايضاً سنستخدم ثلاث ازرار الأول يقوم بعمل emit بالقيمة false لخاصية الكائن كأننا نقوم قم بالإيقاف المؤقت للعداد، اما الزر الثاني فنقوم بعمل emit بالقيمة true لخاصية الكائن وكأننا نقول قم بتشغيل العداد مرة أخرى، اما الزر الأخير فنقوم بتغيير الخاصية الأخرى للكائن بحيث نجعل القيمة الحالية تساوي صفر وكأننا نقول قم بتصفير العداد.

وبعد هذا السرد النظري لنقم باستعراض الشفرة البرمجية والتعليق على اهم النقاط، كالتالي:

ملف app.component.html

```
<div class="text-center p-2">
  <div class="w-75 m-2 fs-1 fw-bolder" #div></div>
</div>
<div>
  <button (click)="pause()">Pause</button>
  <button (click)="reset()">Reset</button>
  <button (click)="start()">Start</button>
</div>
```

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { BehaviorSubject, EMPTY, interval } from 'rxjs';
import { scan, switchMap, tap } from 'rxjs/operators';

export interface IData {
  pause?: boolean;
  currentValue?: number;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('div', { static: true }) private div: ElementRef;

  private counter: BehaviorSubject<IData> = new BehaviorSubject({
    pause: false,
    currentValue: 0,
  });

  constructor() {}

  ngOnInit(): void {
    this.counter
```

```
      .pipe(
        scan((acc: IData, val: IData) => {
          return { ...acc, ...val };
        } , {}),
        switchMap((state: IData) => {
          return state.pause
            ? EMPTY
            : interval(1000).pipe(
                tap(() => {
                  state.currentValue++;
                  this.div.nativeElement.innerHTML = state.currentValue;
                })
              );
        })
      )
      .subscribe();
  }

  start(): void {
    this.counter.next({ pause: false });
  }

  pause(): void {
    this.counter.next({ pause: true });
  }

  reset(): void {
    this.counter.next({ currentValue: 0 });
  }
}
```

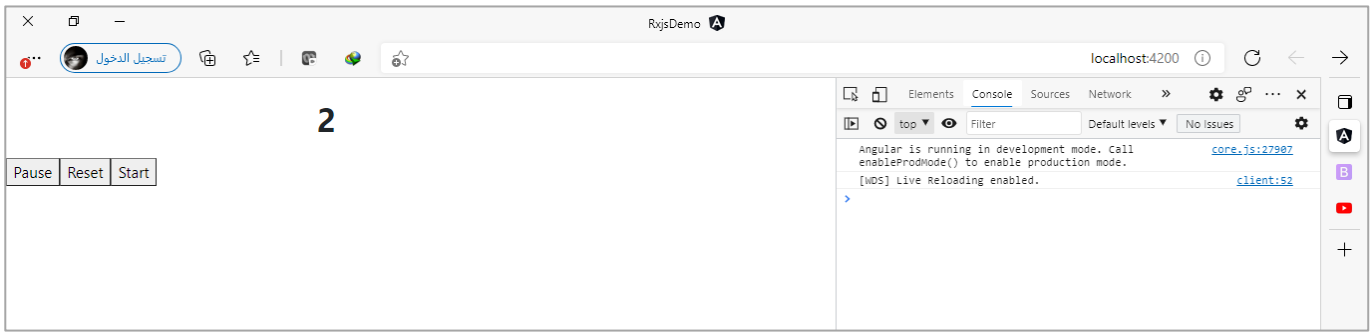
كما نلاحظ استخدمنا الدالة scan لدمج حالات العداد ككائن وليس قيمة العداد، بمعنى أننا استخدمنا الدالة scan لكي تُراقب حالة العداد هل هو pause او reset او start وليس قيمة العداد الذي هو يبدأ 1 ومن ثم 2 ...الخ.

لذلك عندما يتم الضغط على زر pause فسيتم عمل emit للقيمة true وعندما تمر على الدالة scan فسيقوم بدمج القيمة السابقة وهي قيمة pause تساوي false والخاصية الثانية currentValue فتحتوي على قيمة العداد الذي وصل له، وبعدما تم الدمج سوف سيتم ارسال القيمة الجديدة لدالة switchMap والسبب اننا استخدمنا هذه الدالة لكي نُلغي أي Observable سابق ونتعامل مع الObservable الجديد عند تغيير أي قيمة من قيم خصائص الكائن وايضاً لأننا نتعامل مع Observables متداخلة، واخيراً قمنا بعمل شرط لتأكد هل الخاصية pause تساوي true وإذا كانت تساوي true فقم بإرجاع EMPTY أي قيمة فارغة وهذا معناه انه لا يوجد أي قيمة جديدة للعداد، وسيتم إيقاف العداد عند آخر قيمة تم عمل emit لها، اما في حالة كانت false فهذا معناه قراءة العداد ولكي نعمل emit لقيم العداد استخدمنا الدالة interval ومن ثم مررنا القيمة لعنصر HTML الذي اسميناه div.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:

وفي حالة pause:

وأخيراً في حالة reset:



3-1-2- mergeScan():

هذه الدالة تعتبر أقل استخداماً من scan ولكن لها استخداماتها، ونستطيع ان نقول انها مشابهة لعلاقة mergeAll مع map، حيث لو كان لدينا حالة Higher Order Observable مع الدالة scan، فعندئذً نستطيع ان نستخدم scan مع mergeAll او اختصاراً نستخدم الدالة mergeScan.

ولتوضيح لنعطي المثال التالي:

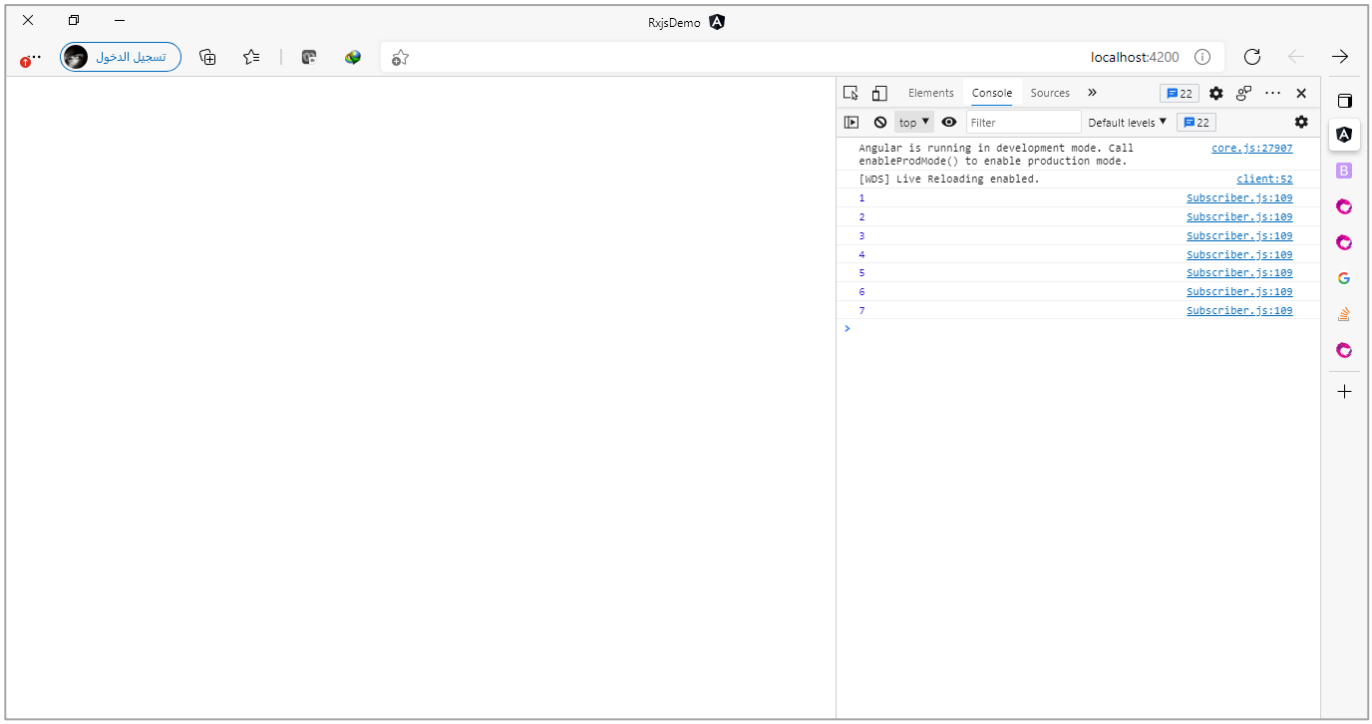
```
import { Component, OnInit } from '@angular/core';
import { fromEvent, of } from 'rxjs';
import { mapTo, mergeScan } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    fromEvent(document, 'click')
      .pipe(
        mapTo(1),
        mergeScan((acc: number, one: number) => {
          return of(acc + one);
        }, 0)
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم قمنا بقراءة كل ضغطة من ضغطات زر الفأرة الأيسر على صفحة التطبيق على شكل Observable ومن ثم مع كل ضغطة قمنا بتحويلها إلى قيمة رقمية وهي واحد 1 ومن ثم استخدمنا الدالة mergeScan لكي نجمع 1 والذي يمثله البارامتر one مع المجمع والذي يمثله المجمع acc وقيمته صفر، ومن ثم أرجعنا Observable وبما أننا أرجعنا Observable فهذه الحالة أصبح لدينا Observables متداخلة لذلك استخدمنا mergeScan لتعامل مع Higher Order Observable.

وعند الضغط مرة أخرى سيتم ارجاع نفس الخطوات السابقة حيث سيتم قراءة الضغطة وتحويلها إلى قيمة رقمية تساوي واحد 1 ومن ثم نجمع القيمة 1 والقادمة من الدالة mapTo مع القيمة السابقة والمخزنة في المجمع acc والذي هو ايضاً قيمته واحد بحيث تكون النتيجة 2 واخيراً نُعيد هذه القيمة على شكل Observable.

وهكذا تتم جميع الخطوات مع كل ضغطة من المستخدم على بزر الفأرة على صفحة التطبيق.



switchScan() -3-3-1-3:

وهذه الدالة مشابهه لدالة mergeScan ولكن هنا مع الاستفادة من مميزات دالة switchAll التي تكلمنا عنها سابقاً في جُزء Higher Order Mapping، مع العلم انها من الدوال الجديدة التي تمت اضافتها في الإصدار السابع من مكتبة rxjs، ومن هذا المنطلق وإلى لحظة كتابة هذا الكتاب لم اجد حالة احتجت فيها إلى استخدام هذه الدالة او وجدت مثال جيد ووضح مهمة هذه الدالة بشكل جيد حتى في الموقع الرسمي لمكتبة rxjs حيث لا يوجد مثال جيد وواضح للفائدة من هذه الدالة على وجه الخصوص، لذلك سنكتفي بما قيل عنها سابقاً بدون ان نورد أي مثال.

Buffering Operators -4-1-3:

والمقصود بها الدوال التالية:

- buffer()
- bufferCount()
- bufferTime()
- bufferWhen()
- bufferToggle()

وهذه الدوال مع انها قليلة الاستخدام وأقل شهرة من دوال Mapping وScanning بجميع أنواعها، ولكن لها استخداماتها التي قد تحتاجها عزيزي المتعلم في أحد تطبيقاتك، لذلك أحببت أن اضيفها ولو على عُجالة.

وتقوم وظيفتها بكل بساطة في التخزين المؤقت Buffering للObservable عند انقطاع Subscription لأي سبب كان وعند رجوع الاتصال مرة أخرى يتم عمل emit للبيانات عند آخر قيمة عند بداية انقطاع الاتصال، ويتم عمل emit لهذه القيم على شكل مصفوفة Arrays.

لذلك هي جميعاً تشترك بهذه الوظيفة، بينما تختلف متى؟ وأين؟ وعدد؟ وهل يتم تحديد بداية ونهاية التخزين ام لا؟

وهذه الأسئلة نُجيب عليها من خلال استعراضنا لهذه الدوال.

buffer() -1-4-1-3:

وهذه الدالة لا تتقيد بأي وقت او عدد وليس هنالك تحديد لبداية ونهاية تخزين القيم، وهي بذلك تعتبر أكثر الدوال مرونة.

وهي تستقبل Observable (نستطيع مجازاً تسميته بالمتحكم) بحيث عندما يتم عمل emit له تقوم هذه الدالة بعمل emit للقيم التي قامت بتخزينها مؤقتاً إلى Subscriber لكي يتم قرائتها، ومن ثم ترجع وتقوم بتخزين قيم أخرى مع حذف القيم السابقة وتنتظر الObservable المتحكم وعندما يعمل emit تقوم هي الأخرى بعمل emit للقيم الموجودة لديها وهكذا.

ولتوضيح لنعطي المثال التالي، والذي نقوم فيه بتوضيح وظيفة الدالة buffer من خلال عمل emit لمجموعة من القيم على شكل Observable باستخدام الدالة interval حيث ستكون القيم على هيئة أرقام وكل ثانية نعمل emit لقيمة.

ولكن لن نعمل لها subscribe لكي نعرض البيانات لأنه كما هو معروف ان الObservable لكي نقرأ بياناته لابد ان نعمل subscribe، وعوضاً عن ذلك سنقوم بقراءة البيانات عن طريق الدالة اسمها pipe حيث سنتكلم عن هذه الدالة لاحقاً وهنا فقط اعرف انني استخدمتها لكي نقرأ أي قيمة يتم عمل لها emit من هذا الObservable.

ولكيلا تضيع هذه القيم نقوم بتخزينها مؤقتاً باستخدام الدالة buffer، وأخيراً عند الضغط على زر معين (وهو عبارة عن المتحكم الذي يخبر هذه الدالة بإن تقوم بعمل emit للقيم التي لديها) نجلب القيم التي تم تخزينها مؤقتاً على شكل مصفوفة، ولكي نحول الحدث click إلى Observable لكي نمرره لدالة buffer استخدمنا الدالة fromEvent.

وأخيراً احببت أن أُشير إلى ان الدالة buffer بعدما تقوم بعمل emit للقيم المخزنة مؤقتاً لديها يتم حذفها مباشرة وتستقبل قيم جديدة، وهكذا إلى أن تنتهي من جميع القيم، كالتالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { buffer, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('btn', { static: true }) private btn: ElementRef;

  value = 0;
  bufferedValues: number[];

  ngOnInit(): void {
    const clicks = fromEvent(this.btn.nativeElement, 'click');
    interval(1000)
      .pipe(
        tap((value: number) => {
          this.value = value;
        }),
        buffer(clicks)
      )
      .subscribe((values) => (this.bufferedValues = values));
  }
}
```

نلاحظ اننا قمنا بعمل buffer للحدث click بعد تحويله إلى Observable عن طريق الدالة fromEvent وهو الذي أطلقنا عليه مجازاً المتحكم، بحيث كلما يتم الضغط على هذا الزر تقوم الدالة buffer بعمل emit للقيم التي قامت بتخزينها مؤقتاً.

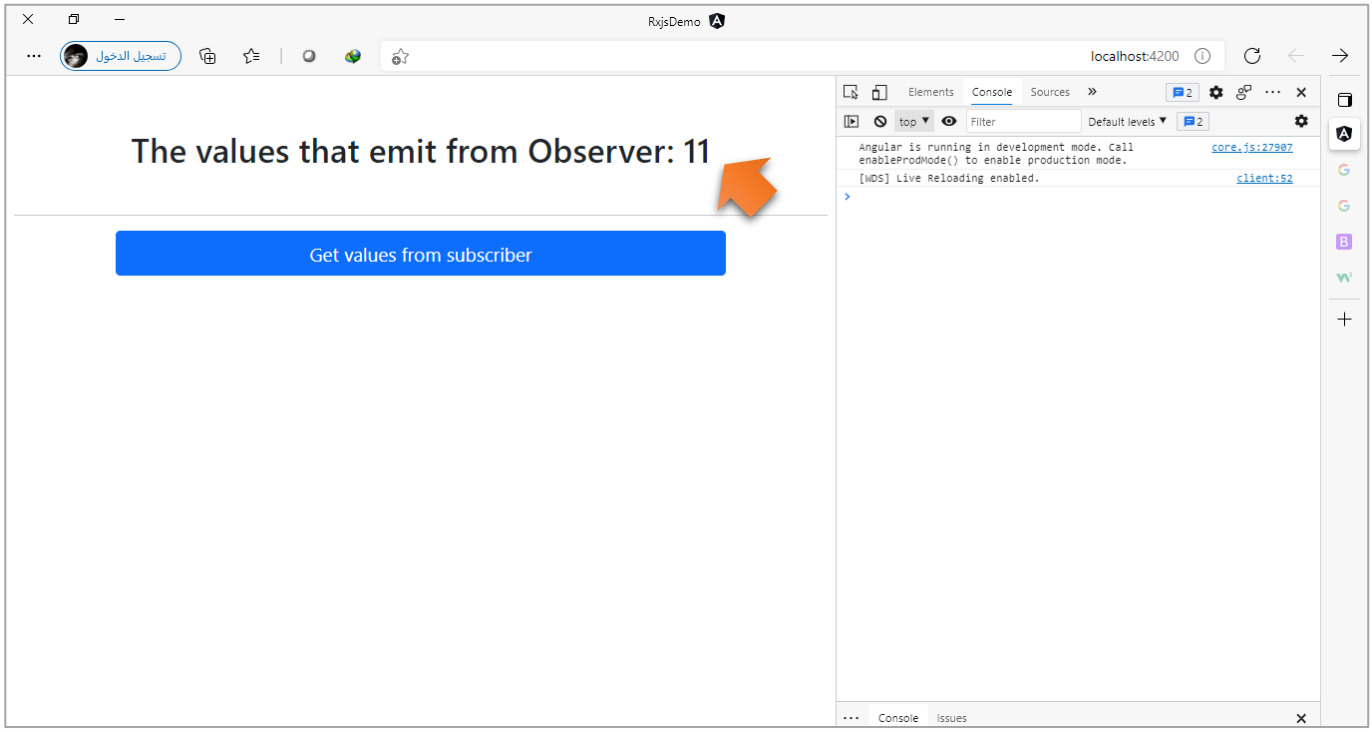
وفي ملف template لهذا component نضيف markup التالي:

ملف app.component.html

```
<div class="text-center p-2">
  <h1 class="m-5">The values that emit from Observer: {{ value }}</h1>
  <hr />
  <button class="btn btn-primary btn-lg w-75" #btn>
    Get values from subscriber
  </button>
  <h1 class="m-5 text-break" *ngIf="bufferedValues">
    [<span class="text-primary">{{ bufferedValues }}</span
    >]
  </h1>
</div>
```

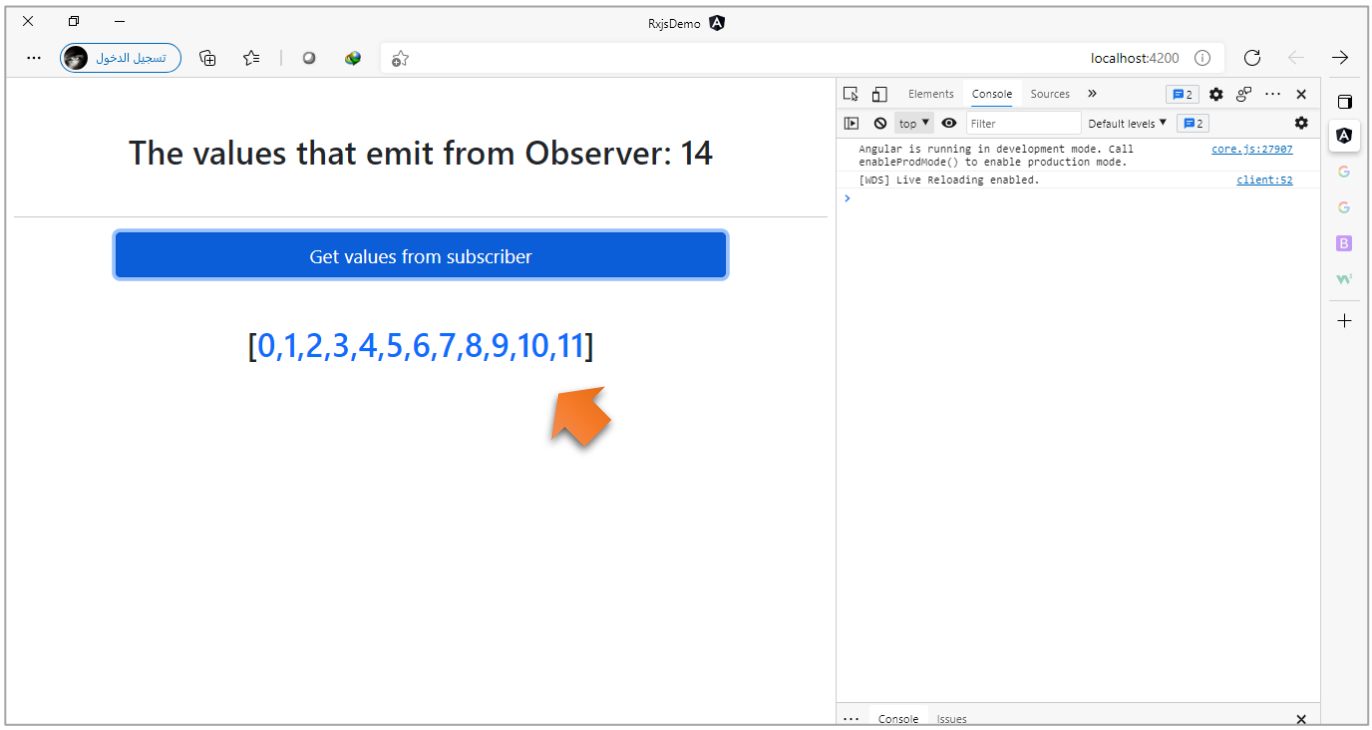
نلاحظ اننا استفدنا من بعض كلاسات مكتبة Bootstrap، بغرض التنسيق فقط.

والآن لنشاهد النتيجة في المتصفح، كالتالي:



كما تلاحظ عزيزي المتعلم جميع القيم التي تم التأشير عليها في السهم بالأعلى هي قيم تم عمل لها emit من الدالة interval وتم تخزينها مؤقتاً في الدالة buffer، وهي إلى الآن لم يتم ارسالها إلى Subscriber لكي نقرئها ونعرضها للمستخدم، وتنتظر

من المتحكم – ان صح التعبير – ان يخبر هذه الدالة بإن تقوم بعمل emit للقيم المخزنة لديها، لذلك لنقوم بالضغط على الزر ولنشاهد النتيجة، كالتالي:



نلاحظ تم جلب القيم وعرضها من خلال Subscriber.

وتقوم بعدها هذه الدالة بحذف القيم المخزنة لديها لاستقبال قيم أخرى، كما أشرنا سابقاً.

3-1-4-2- bufferCount():

هذه الدالة تتيح لنا ان نحدد عدد القيم التي سوف نقوم بعمل تخزين مؤقت لها، فمثلاً لو قمنا بتطبيق المثال السابق ولكن باستخدام الدالة bufferCount واردنا الحصول على آخر قيمتين أي بصيغة أخرى نُريد ان نُخزن آخر قيمتين مؤقتاً، فإن التعديلات على ملف class تكون كالتالي:

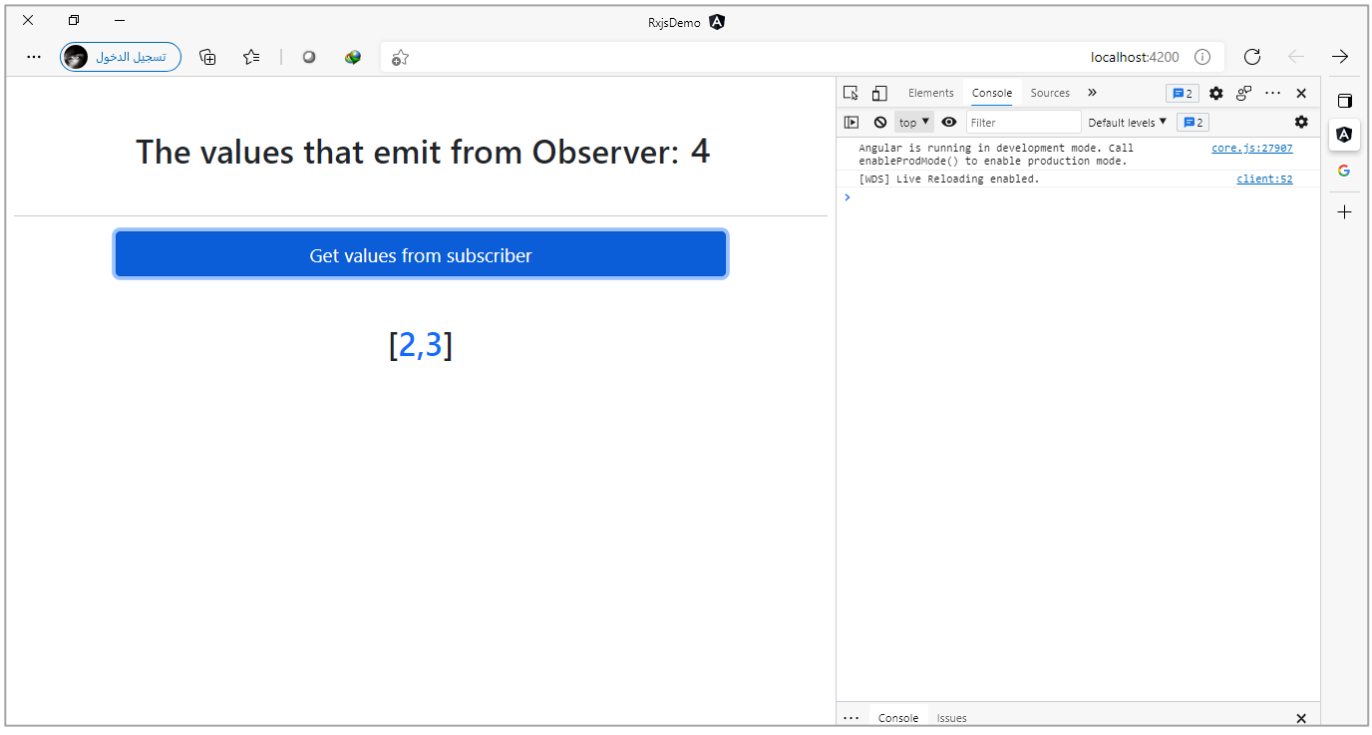
ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { tap, buffer, bufferCount, switchAll } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  @ViewChild('btn', { static: true }) private btn: ElementRef;

  value = 0;
  bufferedValues: number[];
```

```
  ngOnInit(): void {
    const clicks = fromEvent(this.btn.nativeElement, 'click');
    interval(1000)
      .pipe(
        tap((value: number) => {
          this.value = value;
        }),
        buffer(clicks),
        switchAll(),
        bufferCount(2)
      )
      .subscribe((values) => (this.bufferedValues = values));
  }
}
```

والسبب انني استخدمت switchAll، لأن الدالة buffer تُعيد return مصفوفة array وايضاً الدالة bufferCount هي الأخرى تُعيد مصفوفة، لذلك الObservable الذي سيتم قراءته في subscriber ستكون قيمه عبارة عن مصفوفة متداخلة او بمسمى آخر مصفوفة ذات بُعدين، وهذا غير المتوقع لأن المتوقع هو Observable على شكل مصفوفة من القيم لذلك استخدمت الدالة switchAll لكي اعمل flatten للمصفوفة التي تنتج من الدالة buffer على شكل قيم منفردة ومن ثم الدالة bufferCount تعيد تخزين آخر قيمتين بشكل مؤقت على شكل مصفوفة مرة أخرى، وبالطبع تستطيع استخدام mergeAll او concatAll او حتى mergeMap وأخواتها لأن جميع هذه الدوال تشترك بوظيفة عمل flatting لقيم الObservable كما أشرنا في الجزء الخاص الذي شرحنا فيه هذه الدوال.

اما الآن لنقوم بمشاهدة النتيجة في المتصفح، كالتالي:



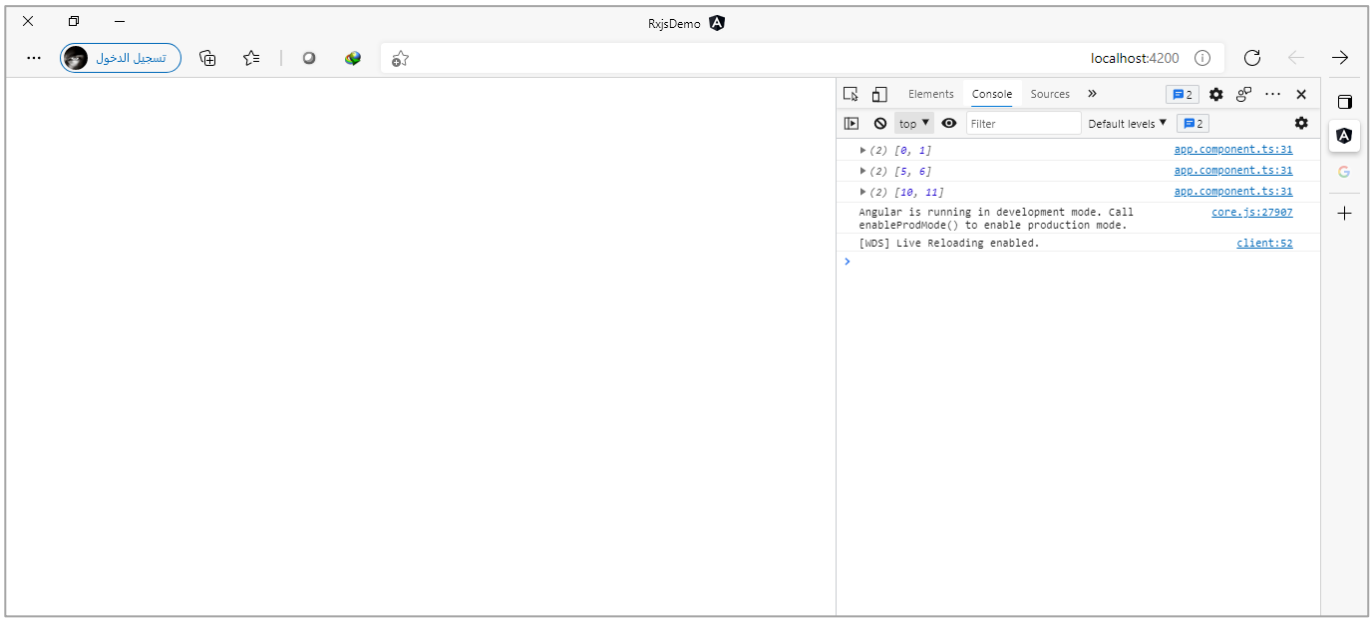
وهنالك باراميتر آخر تستقبله هذه الدالة وهو اختياري، ويُتيح لنا التحكم متى يبدأ تخزين القيم، بصيغة أخرى نستطيع ان نقول انه يعمل عمل المتحكم في الدالة buffer، بحيث نحدد بعد كم قيمة يقوم بتخزين القيم.

ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { of } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  ngOnInit(): void {
    of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
      .pipe(bufferCount(2, 5))
      .subscribe((val) => console.log(val));
  }
}
```

نلاحظ لدينا Observable يحتوي على قيم من 0 إلى 11 واستخدمنا الدالة bufferCount ومررنا لها بارامتران، الأول قيمته 2 ونُشير فيه إلى عدد القيم التي ستقوم هذه الدالة بتخزينها مؤقتاً، والبارامتر الثاني يمثل المُتحكم بحيث يقرر كم قيمة يتم تخطيها، وقد مررنا له القيمة 5 أي قم في البداية بتخزين اول قيمتين بشكل مؤقت وبعدما يتم لهم قراءة عن طريق subscriber قم بالقفز او تخطى 5 قيم وايضاً قم بتخزين قيمتين وهكذا إلى ان ينتهي من جميع القيم، بحيث تكون النتيجة، كالتالي:



3-4-1-3- bufferTime() :

هذه الدالة مشابهة لدالة buffer والفرق الجوهري ان المتحكم – ان صحة التسمية – عبارة عن وقت time وليس Observable كما هو الحال في الدالة buffer.

ويتم تمرير هذا الوقت على شكل ملي ثانية، حيث 1000 ميلي ثانية تُعادل ثانية واحدة.

ولتوضيح لنعطي المثال التالي، حيث سنمرر 5000 ميلي ثانية للدالة bufferTime، بحيث كل خمس ثواني تقوم هذه الدالة بعمل emit لمجموعة من القيم العشوائية المُخزنة مؤقتاً لديها ومن ثم تحذف هذه القيم وتستقبل قيم أخرى وعند مرور خمس ثواني أخرى تعمل emit للقيم الجديدة المُخزنة مؤقتاً ومن ثم تُعيد تكرار الخطوات السابقة، كالتالي:

ملف app.component.html

```
<div class="text-center p-2">
  <h1 class="m-5">The values that emit from Observer: {{ value }}</h1>
  <hr />
  <!-- <button class="btn btn-primary btn-lg w-75" #btn>
    Get values from subscriber
  </button> -->
  <h1 class="m-5 text-break" *ngIf="bufferedValues">
    [<span class="text-primary">{{ bufferedValues }}</span
    >]
  </h1>
</div>
```

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent, interval, of } from 'rxjs';
import { tap, buffer, bufferTime, switchAll, map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('btn', { static: true }) private btn: ElementRef;

  value = 0;
  bufferedValues: number[];

  ngOnInit(): void {
    // const clicks = fromEvent(this.btn.nativeElement, 'click');
    interval(1000)
      .pipe(
        tap((value: number) => {
          this.value = value + 1;
        }),
```

```
        map(() => Math.floor(Math.random() * 100)),
        // buffer(clicks),
        // switchAll(),
        bufferTime(5000)
      )
      .subscribe((values) => (this.bufferedValues = values));
  }
}
```

نلاحظ استخدمنا الدالة map لكي نقوم بتعديل على القيم وعمل تحويل لها بحيث تصبح قيم عشوائية بين الصفر و100، ومن ثم باستخدام الدالة bufferTime مررنا القيمة 5000 أي خمس ثواني، وهذا معناه، كأننا نقول قم بتخزين القيم مؤقتاً وكل خمس ثواني اعمل emit لهذه القيم، ومن ثم احذفها، وكرر نفس العملية كل مرة.

والآن لنشاهد النتيجة في المتصفح، كالتالي:

كما نلاحظ كل خمس ثواني يتم عمل emit لقيم جديدة.

bufferWhen() -4-4-1-3:

وهذه الدالة مشابهه لدالة buffer ولعل الفرق الجوهري ان هذه الدالة نمرر المتحكم – ان صحة التسمية – على شكل دالة function وليس Observable كما هو الحال في الدالة buffer.

وهذه الدالة التي يتم تمريرها إلى bufferWhen تُسمى Selector Function.

اما من ناحية السبب في الحاجة لتمرير Selector Function لأننا بعض الأحيان قد نحتاج ان نكتب بعض الشفرات الإضافية لأي سبب كان عند تخزين البيانات بشكل مؤقت.

ولتوضيح لنعطي نفس المثال السابق مع استخدام الدالة bufferWhen بحيث كلما يتم الضغط على زر جلب البيانات التي تم تخزينها مؤقتاً باستخدام الدالة bufferWhen بالإضافة نضيف سطر برمجي نقوم فيه مثلاً بتخزين رقم مجموعة القيم التي تم عمل لها buffer بصيغة أخرى تخزين مؤقت، بمعنى عند الضغط على الزر أول مرة يتم جلب القيم المُخزنة مؤقتاً ويكون رقمها واحد، ومن ثم عند الضغط على الزر مرة أخرى يتم جلب مجموعة البيانات التالية ويكون رقمها اثنان، وهكذا مع كل مرة يتم جلب القيم المخزنة مؤقتاً يتم زيادة العداد بواحد.

وبطبيعة الحال نستطيع تنفيذ هذا الأمر بعدة طرق منها، ان نقوم بتعريف متغير يكون بمثابة عداد ومن ثم في داخل Selector Function نقوم بزيادة هذا العداد بواحد، كالتالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { tap, buffer, bufferCount, switchAll, map, bufferWhen} from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  @ViewChild('btn', { static: true }) private btn: ElementRef;

  value = 0;
  bufferedValues: number[];
  count = -1;

  ngOnInit(): void {
    const clicks = fromEvent(this.btn.nativeElement, 'click');
    interval(1000)
      .pipe(
        tap((value: number) => {
          this.value = value + 1;
        }),
        map(() => Math.floor(Math.random() * 100)),
        // buffer(clicks),
        // switchAll(),
        bufferWhen(() => {
          this.count++;
          return clicks;
        })
      )
      .subscribe((values) => (this.bufferedValues = values));
```

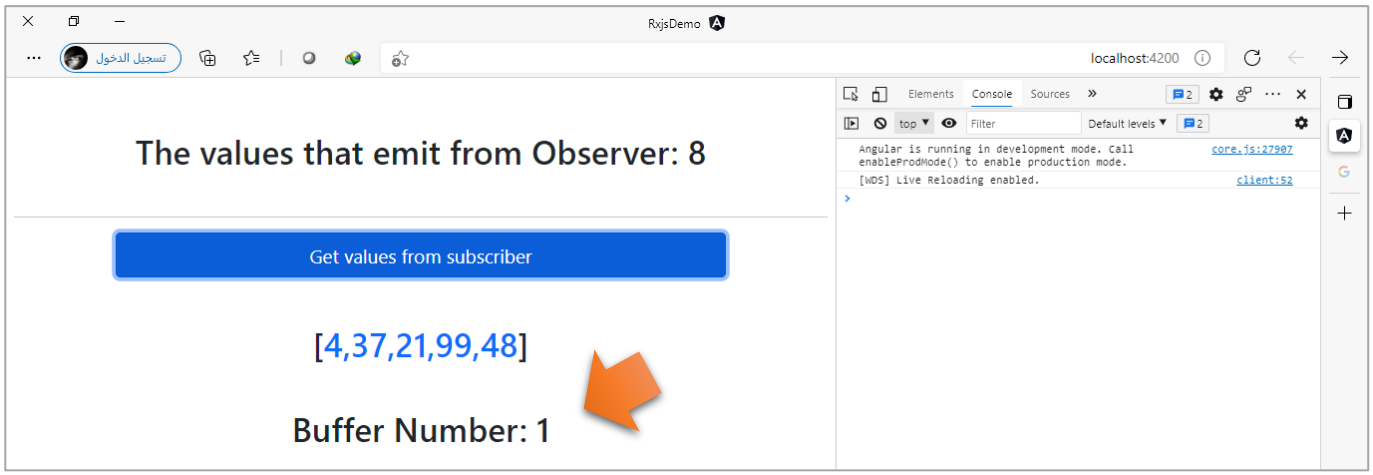
```
  }
}
```

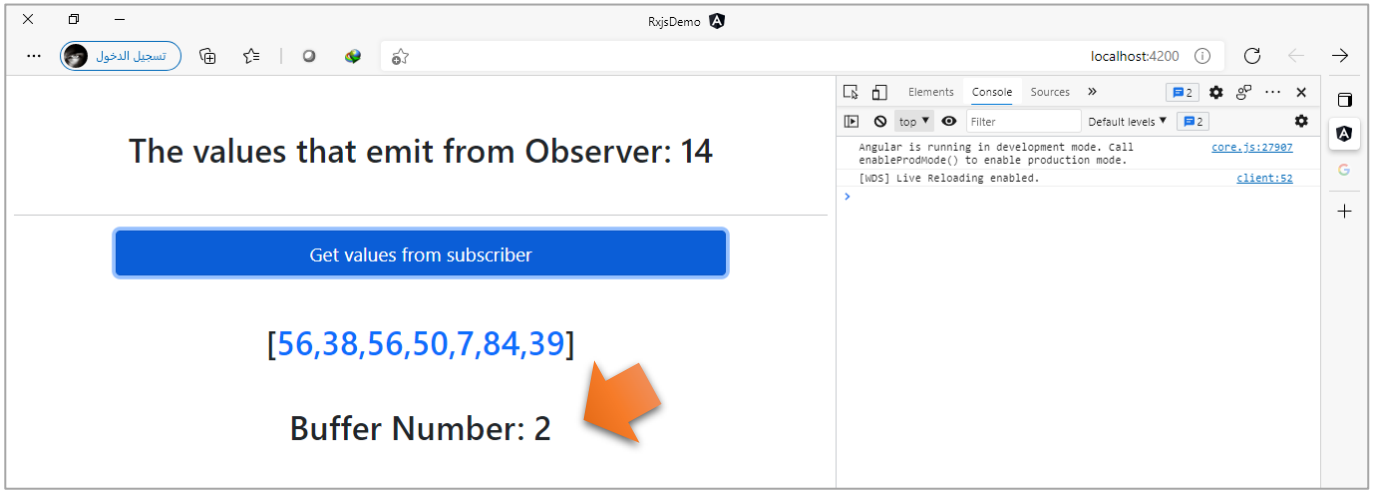
وفي ملف Template نعمل bind للعداد الذي اسميناه count، كالتالي:

ملف app.component.html

```
<div class="text-center p-2">
  <h1 class="m-5">The values that emit from Observer: {{ value }}</h1>
  <hr />
  <button class="btn btn-primary btn-lg w-75" #btn>
    Get values from subscriber
  </button>
  <h1 class="m-5 text-break" *ngIf="bufferedValues">
    [<span class="text-primary">{{ bufferedValues }}</span
    >]
  </h1>
  <h1 class="m-5 text-break" *ngIf="count">Buffer Number: {{ count }}</h1>
</div>
```

والنتيجة في المتصفح تكون، كالتالي:





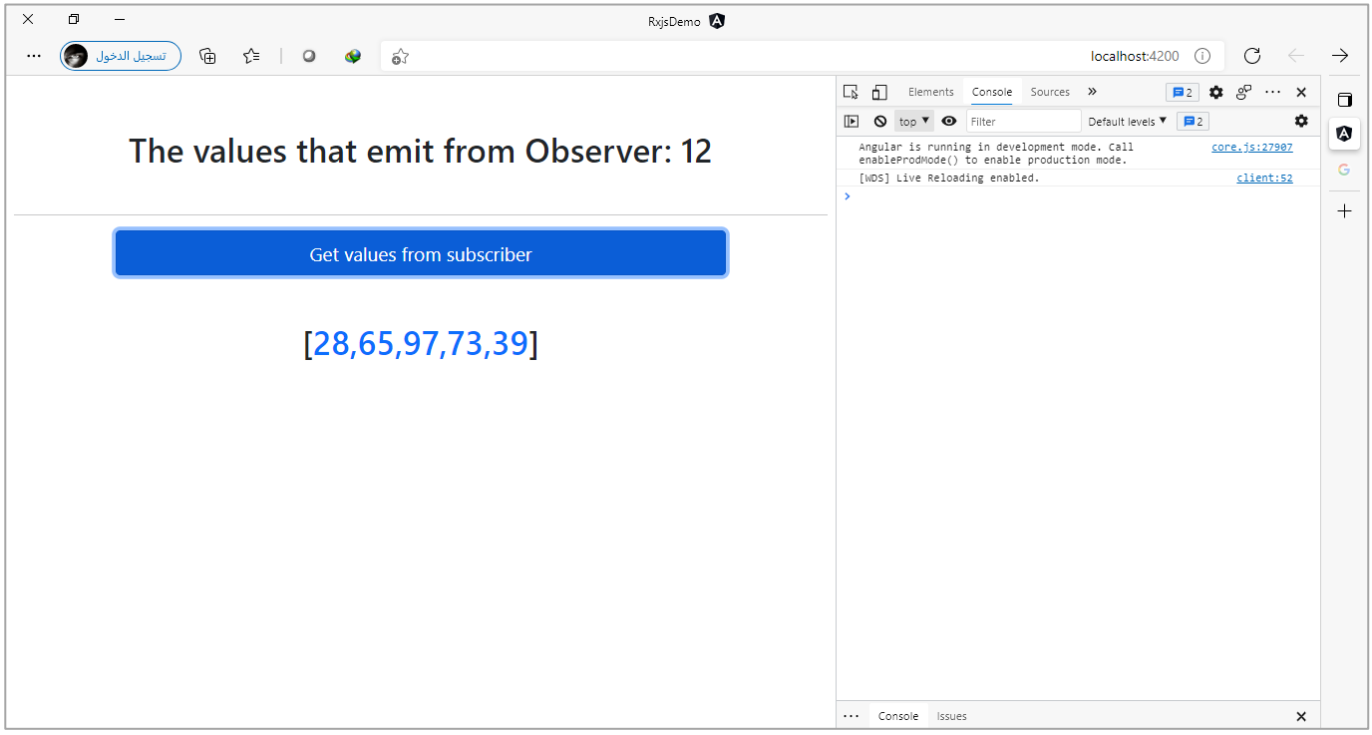
3-1-4-5- bufferToggle():

اما هذه الدالة فلها من اسمها نصيب، حيث تتيح لنا ان نحدد بداية ونهاية buffering، بحيث البارامتر الأول هو الذي يحدد البداية ويكون عبارة عن Observable متشابهاً بذلك مع الدالة buffer، اما البارامتر الثاني فيحدد الفترة الزمنية اللازمة لإيقاف التخزين المؤقت قبل ان يتم ارسالها إلى Subscriber ويكون على شكل دالة function وتُسمى Closer Function، وعند انتهاء الوقت يتم ارسال القيم المخزنة إلى subscriber وايضاً إعادة تخزين القيم مؤقتاً مرة أخرى.

ولتوضيح، لنُعطي المثال التالي حيث سنقوم بإيقاف التخزين لمدة خمس ثواني، كالتالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { tap, map, bufferToggle } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  @ViewChild('btn', { static: true }) private btn: ElementRef;

  value = 0;
  bufferedValues: number[];
  count = -1;

  ngOnInit(): void {
    const clicks = fromEvent(this.btn.nativeElement, 'click');
    interval(1000)
      .pipe(
        tap((value: number) => {
          this.value = value + 1;
        }),
        map(() => Math.floor(Math.random() * 100)),
        bufferToggle(clicks, () => {
          return interval(5000);
        })
      )
      .subscribe((values) => (this.bufferedValues = values));
  }
}
```



Windowing Operators -5-1-3:

المقصود بها الدوال التالية:

- window ()
- windowCount()
- windowTime()
- windowWhen()
- windowToggle()

كما نلاحظ ان الأسماء لهذه الدوال مشابه نوعاً ما لأسماء دوال Buffering، وتقوم وظيفتها بانها تستقبل Observable وتقوم بعمل emit للقيم الخاصة بهذا الObservable على شكل Observable وليس على شكل مصفوفة كما هو الحال في دوال buffering.

وبما انها تُعيد Observable فلابد من استخدام أحدي دوال الHigher Order Mapping كما تعلمنا سابقاً.

كما انها في كل مرة تقوم بعمل emit للقيم المخزنة لديها وفق شرط معين او فترة زمنية معينة – على حسب نوع دالة window – فإنها تقوم بعمل emit لها على شكل مجموعة جديدة وتُسمى هذه المجموعة مجازاً نافذة window، وهكذا كل قيم جديدة يتم انشاء نافذة جديدة لاحتواء هذه القيم (هذه النافذة هي بالأصل عبارة عن Observable جديد ولكن مجازاً يتم تسميتها نافذة window).

اما الآن لنستعرض جميع هذه الدوال على عُجالة، كالتالي:

3-1-5-1- window() :

وهي مشابهه لدالة buffer حيث انها تستقبل بارامتر واحد من النوع Observable وهو المتحكم – ان صحة التسمية – وبناءً عليه يتم انشاء نافذه جديدة تحتوي على مجموعة من القيم.

ولتوضيح لنعطي المثال التالي، حيث فيه يوجد زر وعند الضغط على هذا الزر سيتم انشاء نافذه جديدة تحتوي على مجموعة من القيم، وسوف نستخدم الدالة interval لتوليد القيم والدالة fromEvent ليكون المتحكم من النوع Observable، كالتالي:

ملف app.component.html

```
<div class="text-center p-2">
  <button class="btn btn-primary btn-lg w-75 m-2" #btn (click)="getNewWindow()">
    New Window
  </button>
</div>
```

وفي ملف style لهذا component نضيف الاسطر التالية:

ملف app.component.scss

```
.style-window-element {
  display: inline-block;
  border: 1px solid gray;
  border-radius: 50%;
  margin: 10px;
  background: honeydew;
  padding: 10px;
  width: 50px;
  text-align: center;
}
```

واخيراً نضيف الشفرات البرمجية التالية في ملف class، كالتالي:

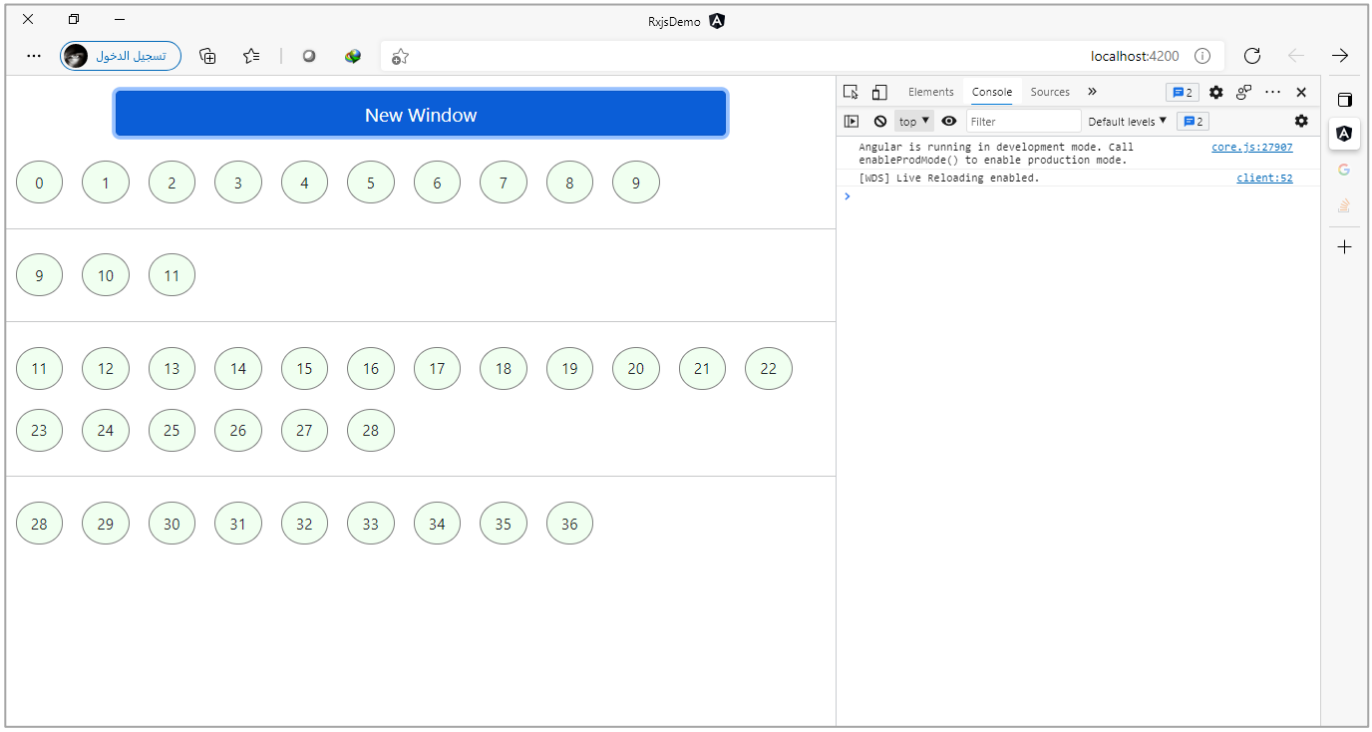
ملف app.component.ts

```
import {
  Component,
  ElementRef,
  OnInit,
  Renderer2,
  ViewChild,
} from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { mergeAll, window } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

```
})
export class AppComponent implements OnInit {
  @ViewChild('btn', { static: true }) private btn: ElementRef;

  value = 0;

  constructor(private element: ElementRef, private renderer: Renderer2) {}

  ngOnInit(): void {
    const click$ = fromEvent(this.btn.nativeElement, 'click');
    interval(1000)
      .pipe(window(click$), mergeAll())
      .subscribe((value) => {
        this.value = value;
        this.createElement(value);
      });
  }

  getNewWindow(): void {
    const hr = this.renderer.createElement('hr');
    this.renderer.appendChild(this.element.nativeElement, hr);
    this.createElement(this.value);
  }

  private createElement(value: number): void {
    const div = this.renderer.createElement('div');
    const text = this.renderer.createText(value.toString());
    this.renderer.addClass(div, 'style-window-element');
    this.renderer.appendChild(div, text);
    this.renderer.appendChild(this.element.nativeElement, div);
  }
}
```

الذي يخص هذه الدالة هو ما تم التأشير عليه بالسهم، اما باقي الاسطر البرمجية فهي فقط لإعطاء شكل رسومي وتوضيحي لك عزيزي المتعلم عند تشغيل التطبيق، وقد تم شرح اغلبها في كتابنا المعنون Angular Component and Services وهو جزء من هذه السلسلة.

والآن لنقوم بتشغيل التطبيق ولنرى النتيجة، كالتالي:



نلاحظ مع كل ضغطة يتم انشاء نافذة جديدة وفيها قيم جديدة.

3-1-5-2- windowCount():

ولها من أسمها نصيب حيث تقوم هذه الدالة بأخذ مجموعة من القيم يتم تمرير عددها لهذه الدالة على شكل بارامتر لكل نافذة window.

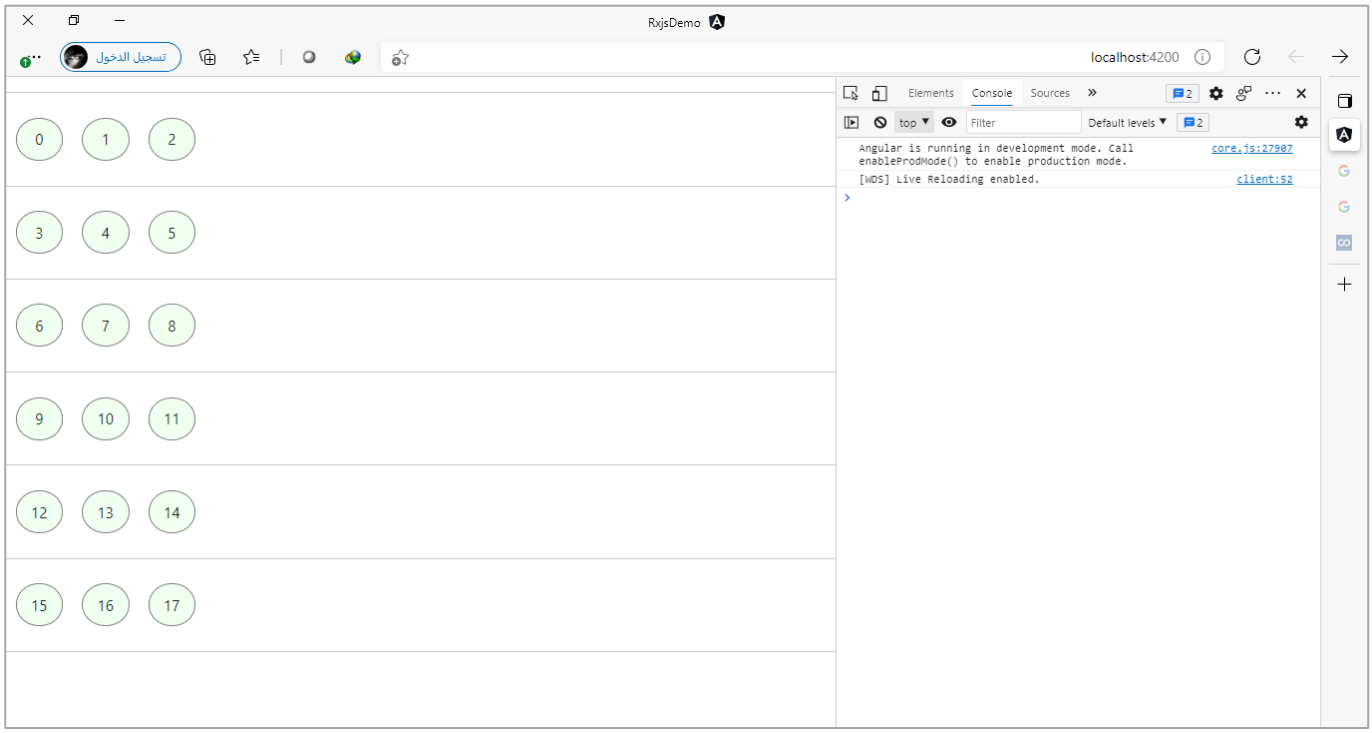
ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, Renderer2, ViewChild,} from '@angular/core';
import { interval } from 'rxjs';
import { mergeAll, windowCount } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor(private element: ElementRef, private renderer: Renderer2) {}

  ngOnInit(): void {
    interval(1000)
      .pipe(
        windowCount(3),
        tap(() => {
          const hr = this.renderer.createElement('hr');
          this.renderer.appendChild(this.element.nativeElement, hr);
```

```
        }),
        mergeAll()
      )
      .subscribe((value) => {
        const div = this.renderer.createElement('div');
        const text = this.renderer.createText(value.toString());
        this.renderer.addClass(div, 'style-window-element');
        this.renderer.appendChild(div, text);
        this.renderer.appendChild(this.element.nativeElement, div);
      });
  }
}
```

ولنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ كل ثلاث قيم يتم عمل emit لها في نافذة جديدة، والسبب لأننا مررنا القيمة 3 لهذه الدالة على شكل بارامتر.

windowTime() -3-5-1-3:

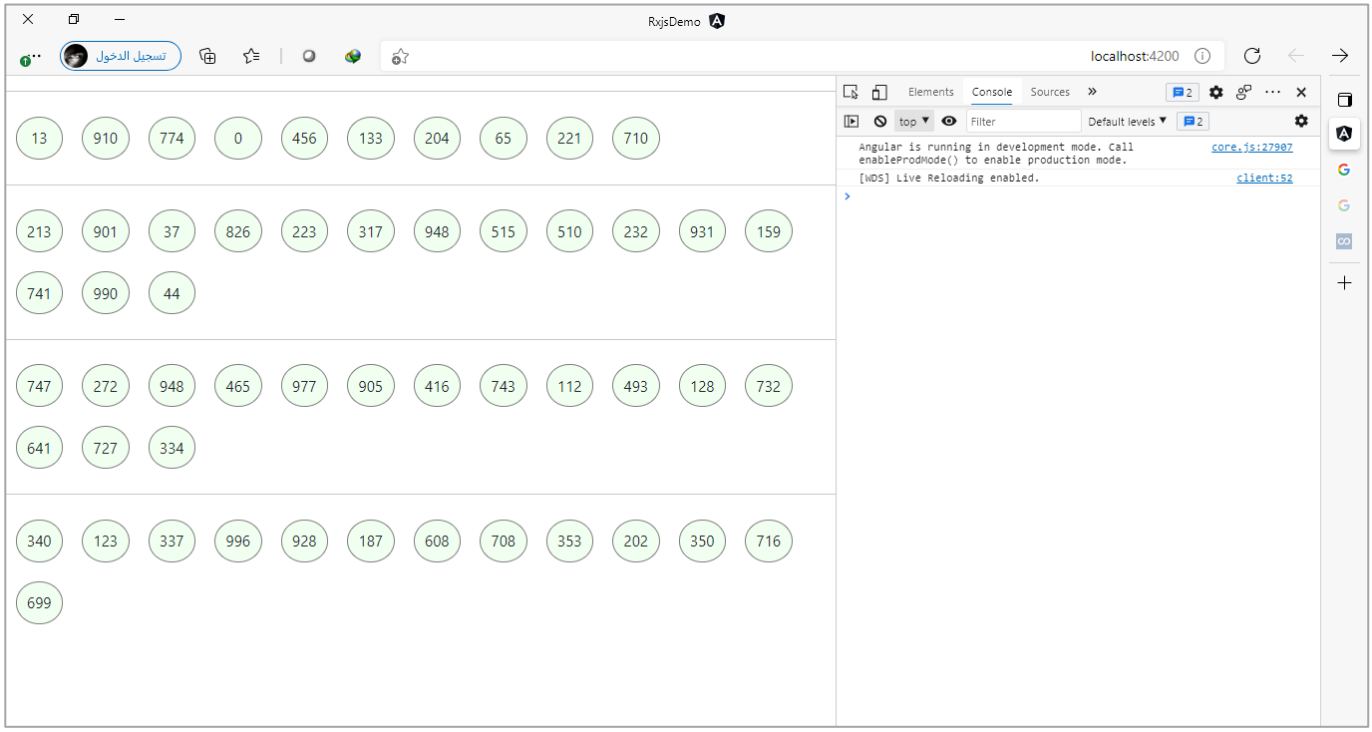
اما هذه الدالة فتقوم بعمل emit للObservable خلال فترة زمنية معينة، وتستقبل بارامتر على شكل رقم يُمثل الوقت بالملي ثانية، وهي تختلف عن الدالة السابقة windowCount حيث هناك كنا نقول نأخذ عدد قيم مشابهه للرقم المُمرر لدالة، اما هنا في الدالة windowTime وقت وتقوم بعمل emit للObservable على شكل نافذة window بحسب الوقت الممرر وعند انتهاء الوقت وهنالك قيم جديدة تقوم بإنشاء نافذة جديدة وعمل emit لهذه القيم على شكل Observable، وهكذا إلى ان تنتهي من جميع القيم.

ولتوضيح لنعطي المثال التالي:

app.component.ts ملف

```
import { Component, ElementRef, OnInit, Renderer2, ViewChild } from '@angular/core';
import { fromEvent, interval, Observable } from 'rxjs';
import { map, mergeAll, tap, windowTime } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private element: ElementRef, private renderer: Renderer2) {}

  ngOnInit(): void {
    interval(Math.floor(Math.random() * 2000))
      .pipe(
        windowTime(5000),
        tap(() => {
          const hr = this.renderer.createElement('hr');
          this.renderer.appendChild(this.element.nativeElement, hr);
        }),
        mergeAll(),
        map(() => Math.floor(Math.random() * 1000))
      )
      .subscribe((value) => {
        const div = this.renderer.createElement('div');
        const text = this.renderer.createText(value.toString());
        this.renderer.addClass(div, 'style-window-element');
        this.renderer.appendChild(div, text);
        this.renderer.appendChild(this.element.nativeElement, div);
      });
  }
}
```

نلاحظ مررنا قيم عشوائية لدالة interval لكي يختلف عدد القيم في كل Observable، ومن ثم في الدالة windowCount مررنا القيمة 5000 أي كل خمس ثواني قم بعمل emit لنافذة جديدة تحتوي على مجموعة من القيم العشوائية، اما الآن لنشاهد النتيجة في المتصفح:



windowWhen() -4-5-1-3:

هذه الدالة مشابهه لدالة window ولكن عوضاً عن استقبال المتحكم – إن صحة التسمية – على شكل Observable يتم استخدامه هنا على شكل دالة، وفي محتوى هذه الدالة نكتب أي شفرة برمجية نُريدها بالإضافة إلى الاسطر البرمجية الخاصة بتحديد متى يتم انشاء نافذة جديدة.

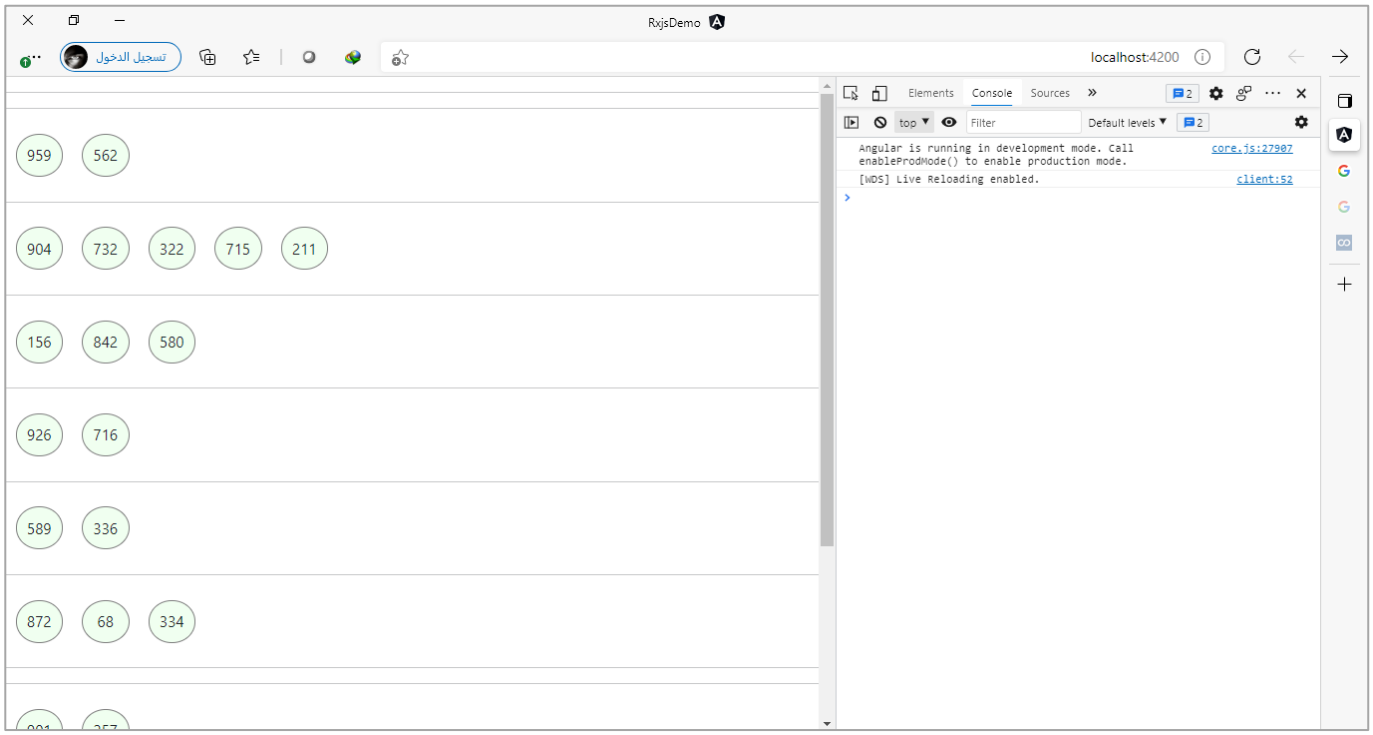
ولتوضيح لنعطي المثال التالي، حيث سنمرر دالة function إلى دالة windowWhen ونضيف وقت عشوائي بحيث عند مرور هذا الوقت قم بفتح نافذة جديدة، كالتالي:

ملف app.component.ts

```
import { Component, ElementRef, OnInit, Renderer2 } from '@angular/core';
import { interval } from 'rxjs';
import { map, mergeAll, tap, windowWhen } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor(private element: ElementRef, private renderer: Renderer2) {}

  ngOnInit(): void {
    interval(1000)
      .pipe(
        windowWhen(() => interval(Math.random() * 4000)),
        tap(() => {
          const hr = this.renderer.createElement('hr');
```

```
          this.renderer.appendChild(this.element.nativeElement, hr);
        }),
        mergeAll(),
        map(() => Math.floor(Math.random() * 1000))
      )
      .subscribe((value) => {
        const div = this.renderer.createElement('div');
        const text = this.renderer.createText(value.toString());
        this.renderer.addClass(div, 'style-window-element');
        this.renderer.appendChild(div, text);
        this.renderer.appendChild(this.element.nativeElement, div);
      });
  }
}
```

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



windowToggle() -5-5-1-3:

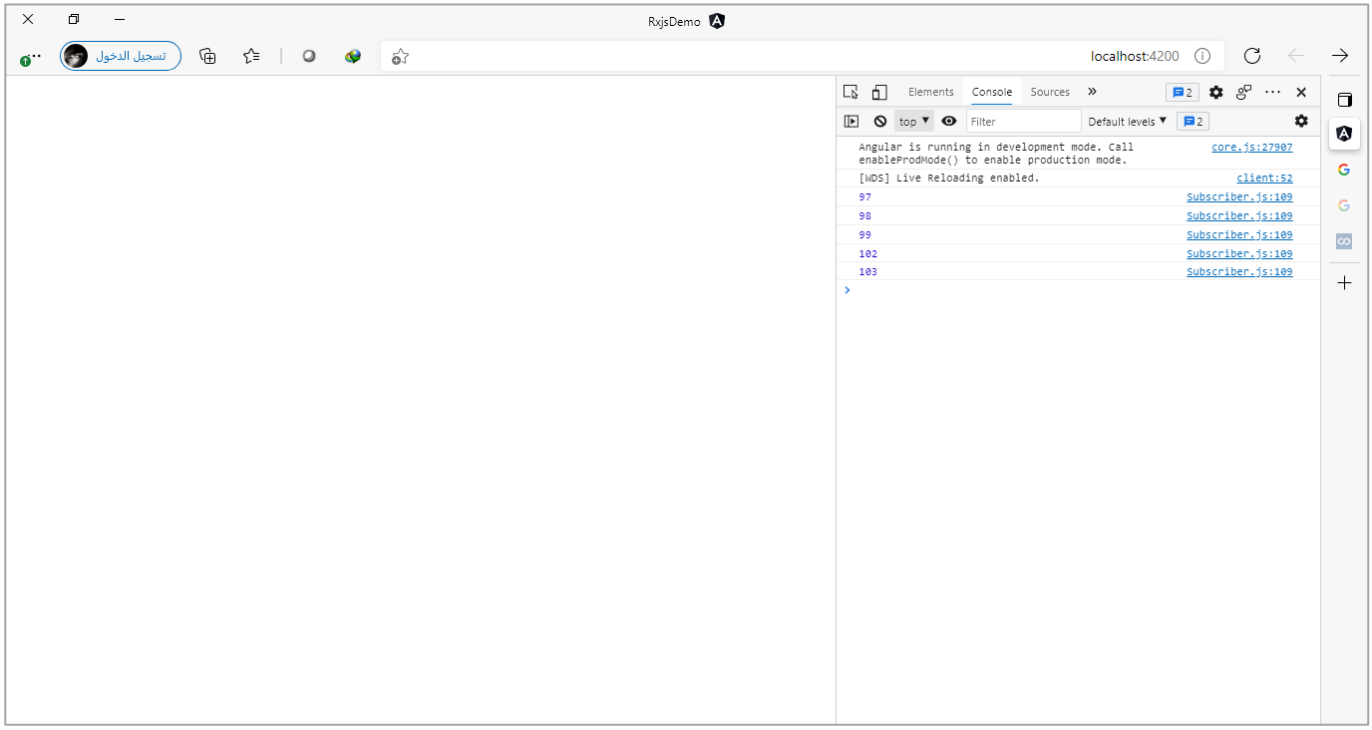
اما هذه الدالة فتستقبل بارامترين الأول يحدد متى يبدأ والثاني على شكل دالة يحدد فترة او الوقت لقراءة القيم وإعادتها بنافذة على شكل Observable.

ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { mergeAll, windowToggle } from 'rxjs/operators';

@Component({
```

```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    const clicks = fromEvent(document, 'click');
    interval(2000)
      .pipe(
        windowToggle(clicks, () => interval(5000)),
        mergeAll()
      )
      .subscribe(console.log);
  }
}
```



:reduce() Operators -6-1-3

هذه الدالة مشابهة ايضاً لدالة reduce في مصفوفات الجافا سكريبت، وايضاً مشابهه لدالة scan والفرق الجوهري هو انها لا تقوم بعمل emit للقيم مع كل عملية دمج وانما بعد الانتهاء من دمج جميع القيم تقوم بعمل emit لناتج النهائي.

وهذه الدالة مثلها مثل الدالة scan لها استخدامات شائعة وهي دمج وجمع البيانات ولها استخدامات اقل شيوعاً وهي اجراء العمليات الحسابية على القيم الرقمية.

اما السؤال المهم وهو متى نستخدم scan ومتى نستخدم reduce؟ والإجابة تعتمد على احتياجك انت عزيزي المتعلم فإذا كنت تُريد فقط الناتج النهائي لعملية دمج البيانات او العملية الحسابية فلعل الدالة reduce هي المناسبة ام إذا كنت تُريد قراءة والتعامل مع كل عملية دمج للبيانات او ناتج كل عملية حسابية فعندئذً يُفضل استخدام الدالة scan.

اما الآن لنستعرض بعض الأمثلة البسيطة على هذا الدالة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { reduce } from 'rxjs/operators';
import { of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(1, 2, 3, 4, 5)
      .pipe(
        reduce((value: number, acc: number) => {
          return value * acc;
        }, 1)
      )
      .subscribe(console.log);
  }
}
```

سوف تلاحظ عزيزي المتعلم ان النتيجة هي حاصل الضرب النهائي لجميع القيم، قم بتغيير الدالة reduce بالدالة scan وشاهد الفرق؟

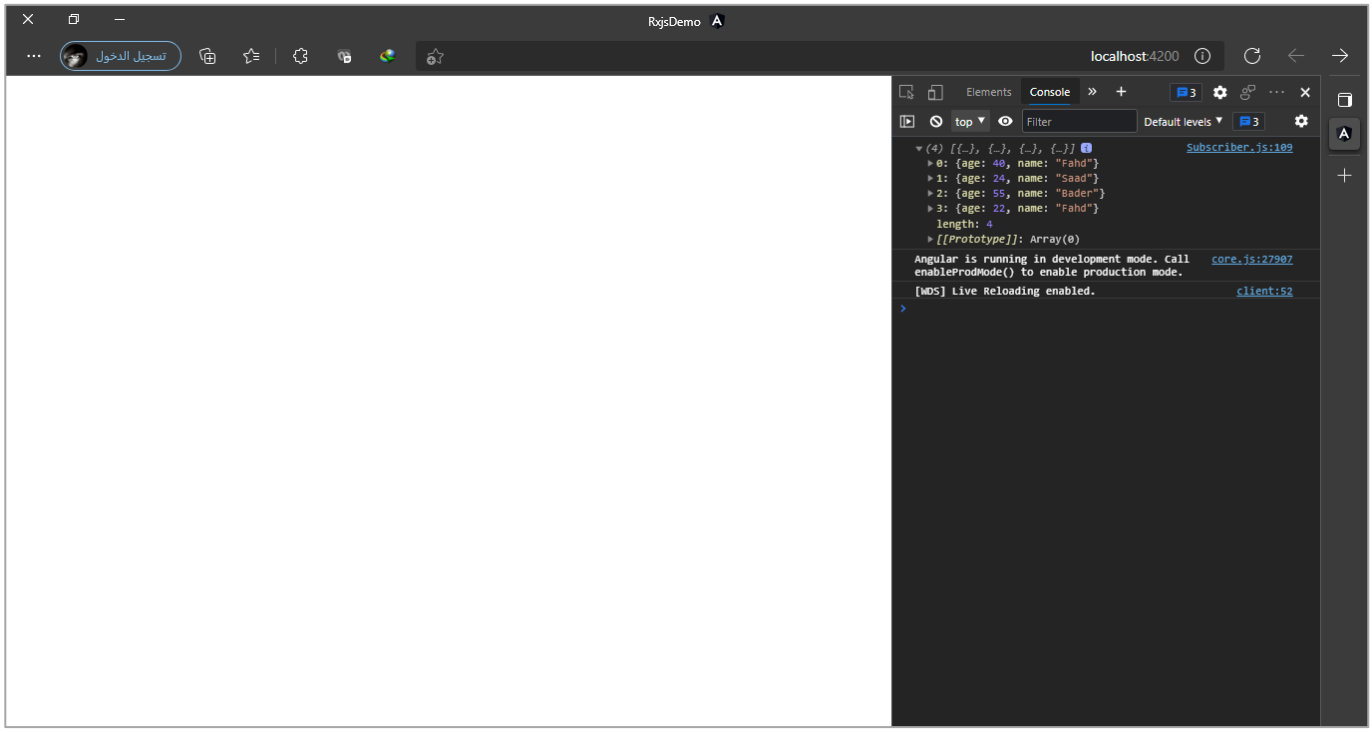
والآن لنستعرض مثال آخر لدمج مجموعة من البيانات وسنستخدم الدالة reduce هذه المرة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { reduce } from 'rxjs/operators';
import { of } from 'rxjs';

export interface Person {
  age: number;
  name: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

```
})

export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(
      { age: 22, name: 'Fahd' },
      { age: 55, name: 'Bader' },
      { age: 24, name: 'Saad' },
      { age: 40, name: 'Fahd' }
    )
      .pipe(
        reduce((value: Person[], acc: Person) => {
          return [acc, ...value];
        }, [])
      )
      .subscribe(console.log);
  }
}
```

وسوف نلاحظ ان الذي يظهر هي نتيجة الدمج النهائية لجميع هذه البيانات، كالتالي:



pairwise() Operator -7-1-3:

لو رجعنا قليلاً وخصوصاً لدالة zip لوجدنا اننا أشرنا ان وظيفة هذه الدالة هي دمج قيم مجموعة من الObservables مع بعضها البعض، بحيث تأخذ اول قيمة من الObservable الأول وأول قيمة من الObservable الثاني وتجمعهما في مصفوفة ومن ثم تعمل emit لهذه المصفوفة، ومن ثم تنتقل للقيمة الثانية من كلا الObservables الأول والثاني وايضاً

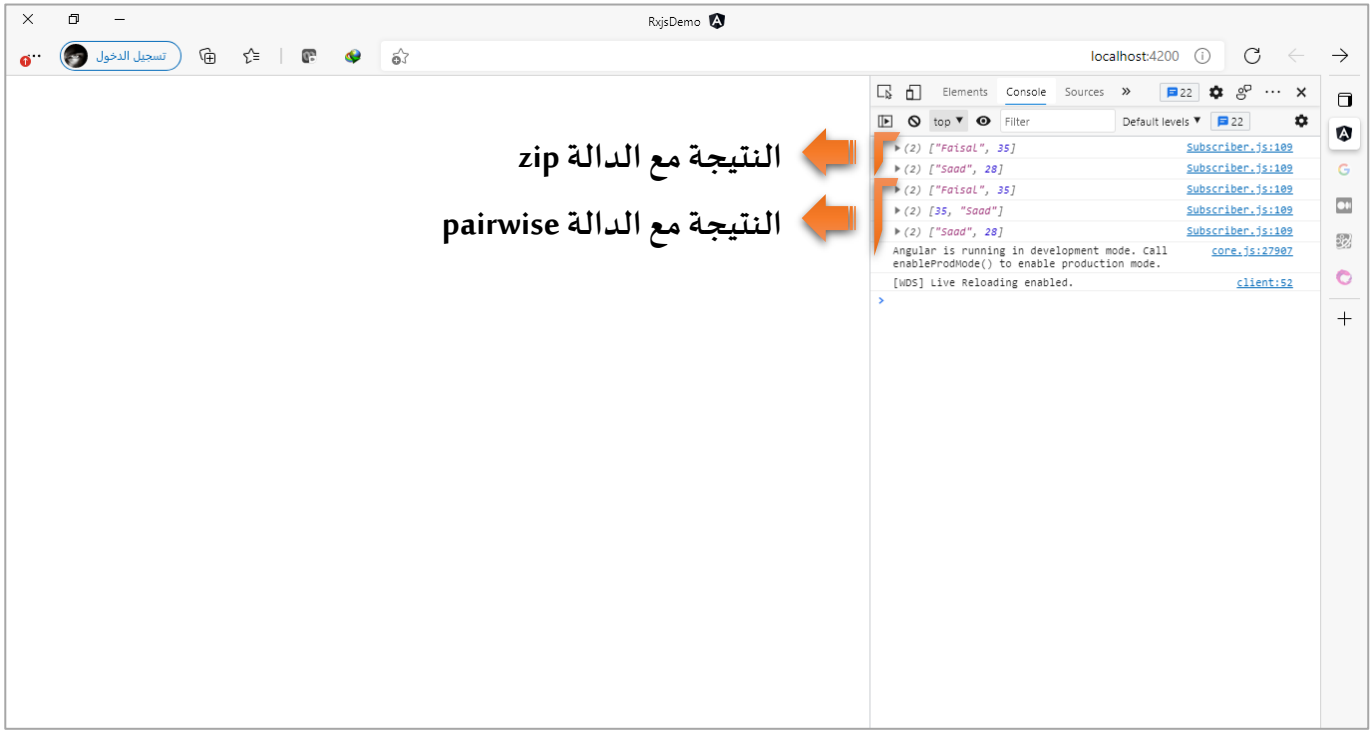
تجمعهما في مصفوفة وتعمل emit لهذه المصفوفة، وهكذا إلى ان تنتهي من جميع القيم ولو كان هنالك قيمة في الObservable الأول ولكن لا يوجد ما يقابله في الObservable الثاني فسيتم اهمال هذه القيمة.

تقريباً مهمة الدالة pairwise مشابهة نوعاً ما لدالة zip، مع وجود بعض الاختلافات البسيطة، اما أوجه الشبه فهي تأخذ كل قيمتين وتجمعهما في مصفوفة ومن ثم تعمل emit لهذه المصفوفة وايضاً لا تعمل emit لأول قيمة إلى تتحصل على القيمة التالية لكي تعمل لهم دمج على شكل مصفوفة، اما أوجه الاختلاف فهذه الدالة تتعامل مع Observable واحد وتعمل دمج لقيمه، وايضاً هي تأخذ القيمة الأولى والثانية ومن ثم الثانية والثالثة وهكذا إلى ان تنتهي من جميع القيم وأخيراً لا توجد أي قيم ممكن ان يتم اهمالها لأنها تأخذ كل قيم والقيمة التي تليها ومن ثم تأخذ القيمة التي تليها مع القيمة التي تلي هذه القيمة الأخيرة وهكذا إلى ان تنتهي من جميع القيم.

ولتوضيح لنعطي المثال التالي:

```
import { Component, OnInit } from '@angular/core';
import { from, zip } from 'rxjs';
import { pairwise } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  ngOnInit(): void {
    const x1 = ['Faisal', 'Saad'];
    const x2 = [35, 28];
    const x3 = ['Faisal', 35, 'Saad', 28];

    zip(from(x1), from(x2)).subscribe(console.log);

    from(x3).pipe(pairwise()).subscribe(console.log);
  }
}
```

والنتيجة تكون كالتالي:



3-1-8- toArray() Operator:

ولها من أسمها نصيب حيث تقوم هذه الدالة بالانتظار إلى ان يكتمل الObservable ومن ثم تعمل emit لجميع القيم على شكل مصفوفة. ولها استخداماتها فمثلاً لو كان لدينا مجموعة من البيانات على شكل مصفوفة ومن ثم احتجنا أن نعمل لها flatten – تجزئة المصفوفة – لأي سبب من الأسباب وبعد الانتهاء من اجراء العمليات الضرورية نُريد أن نُعيد هذه البيانات إلى اصلها وهو على شكل مصفوفة، او إذا اتتنا بيانات مجزئة من الأساس ونُريد ان نجمعها في مصفوفة واحد لكي نُجري عليها بعض العمليات الضرورية.

ولتوضيح لنعطي مثال حيث عن طريقه نفترض انه لدينا مجموعتين من البيانات على شكل مصفوفة من الكائنات الأولى تحتوي على بيانات الشركات المصنعة للسيارات، والثانية تحتوي على بيانات السيارات وكُل سيارة تتضمن من ضمن بياناتها id الخاص بكل شركة مصنعة لسيارة المحددة.

وسوف نقوم بعمل flat لمصفوفة بيانات السيارات ومن ثم نجري بعض العمليات ولتكن على سبيل المثال ان نبحث بهذه البيانات ومن ثم نغير رقم id الخاص بالشركة المصنعة باسم الشركة، واخيراً نعيد تجميع هذه البيانات على شكل مصفوفة، كالتالي:

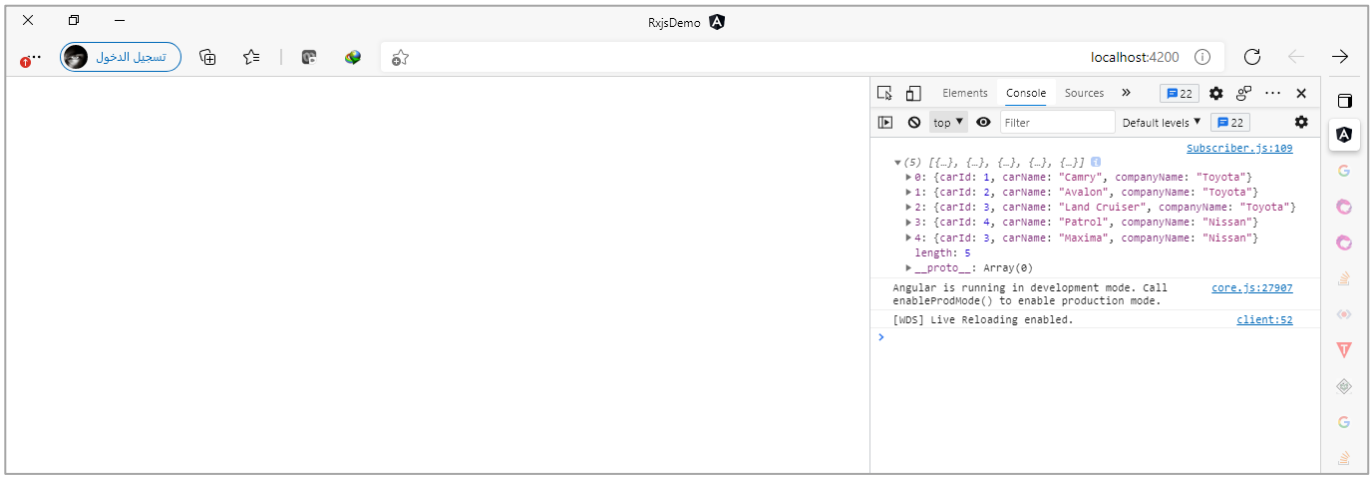
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';
import { map, toArray } from 'rxjs/operators';

export interface Company {
  companyId: number;
  companyName: string;
}
export interface Car {
```

```
  carId: number;
  carName: string;
  companyId: number;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}
  private companies: Company[] = [
    {
      companyId: 1,
      companyName: 'Toyota',
    },
    {
      companyId: 2,
      companyName: 'Nissan',
    },
  ];

  public cars: Car[] = [
    {
      carId: 1,
      carName: 'Camry',
      companyId: 1,
    },
    {
      carId: 2,
      carName: 'Avalon',
      companyId: 1,
    },
    {
      carId: 3,
      carName: 'Land Cruiser',
      companyId: 1,
    },
    {
      carId: 4,
      carName: 'Patrol',
      companyId: 2,
    },
    {
      carId: 3,
      carName: 'Maxima',
      companyId: 2,
    },
  ];
```

```
  ngOnInit(): void {
    from(this.cars)
      .pipe(
        map((car: Car) => {
          const { carId, carName, companyId } = car;
          const companyName = companyId === 1 ? 'Toyota' : 'Nissan';
          return { carId, carName, companyName };
        }),
        toArray()
      )
      .subscribe(console.log);
  }
}
```

كما نلاحظ مصفوفة بيانات الشركات اسميتها companies بينما مصفوفة بيانات السيارات اسميتها cars وايضاً قمت بتعريف اثنين interfaces لكي نضع تعريف ووصف لهذه البيانات، ومن ثم اضفنا بعض البيانات العشوائية والخطوة التي تليها قمنا بعمل flat او تجزئة لمصفوفة cars باستخدام الدالة from، ومن ثم استخدمنا الدالة map لتلاعب وتحويل وتعديل البيانات وأخيراً استخدمنا الدالة toArray لكي نقوم بجمع ودمج البيانات.



3-1-9- Operator ()expand:

هذه الدالة نفس مفهوم Recursion في أغلب لغات البرمجة ومنها لغة الجافا سكريبت والذي هو عبارة عن Design Pattern، ومن تطبيقاته Recursive Function وهي عبارة عن دالة تُعيد نفسها أي تعمل return لنفسها من داخل الدالة.

وفي الحقيقة الكلام بهذا الموضوع شاسع وليس هنا المقام لشرحه وتستطيع عزيزي المتعلم الرجوع إلى العديد من المراجع العربية والأجنبية وبالتحديد تلك التي تتحدث عن Design Patterns للاطلاع بشكل أعمق عن هذا المفهوم والمفاهيم الأخرى.

ولنرجع الآن إلى محور حديثنا وهو دالة expand حيث تستقبل هذه الدالة Observable وتُعيد Observable جديد يحتوي على القيم بعد التعديد إذا كان هنالك تعديل بالطبع، وتقوم هذه الدالة بعمل إعادة لنفسها وتنفيذ الأسطر إلى

مالا نهاية إلى ان نضع شرط بداخلها يوقف هذه الدالة، وبما أنها تُعيد نفسها فلابد أن تكون الإعادة على شكل Observable، وليس قيمة عادية.

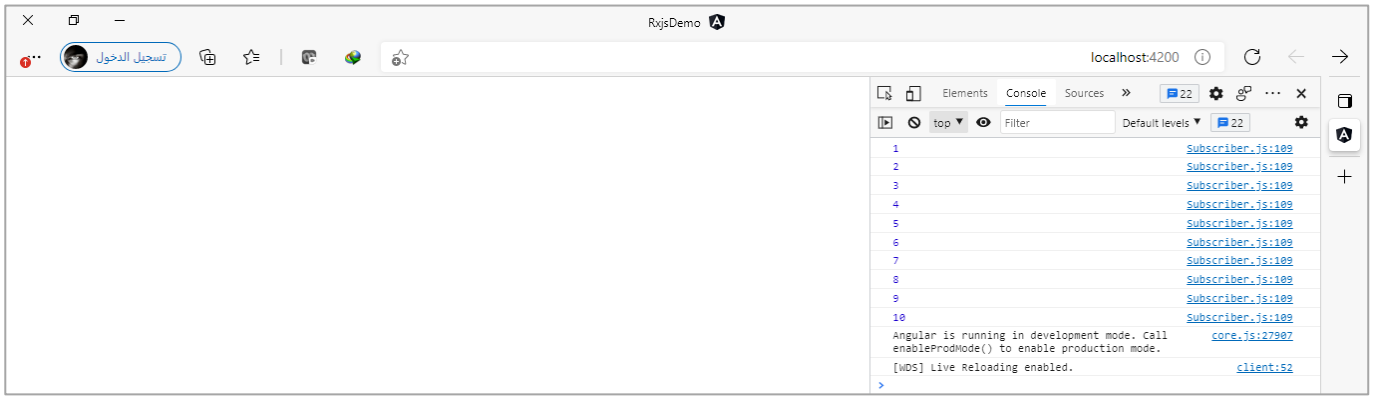
ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { EMPTY, of } from 'rxjs';
import { expand } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    of(1)
      .pipe(
        expand((val: any) => {
          return val === 10 ? EMPTY : of(val + 1);
        })
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم يوجد لدينا Observable يحتوي على قيمة واحدة ومررنا هذه القيمة على دالة expand وهذه الدالة قامت بتكرار نفسها ووضعنا شرط بحيث إذا أصبحت القيمة تساوي عشرة أوقف هذه الدالة وذلك باستخدام الدالة EMPTY وإذا لم يكن اسمع لدالة بتكرار نفسها ولكن نُعيد القيمة مُضاف لها واحد على شكل Observable ويتضح هذا جلياً في الأمر of(val + 1).

اما النتيجة في المتصفح، فتكون كالتالي:



3-1-10- groupBy() Operator:

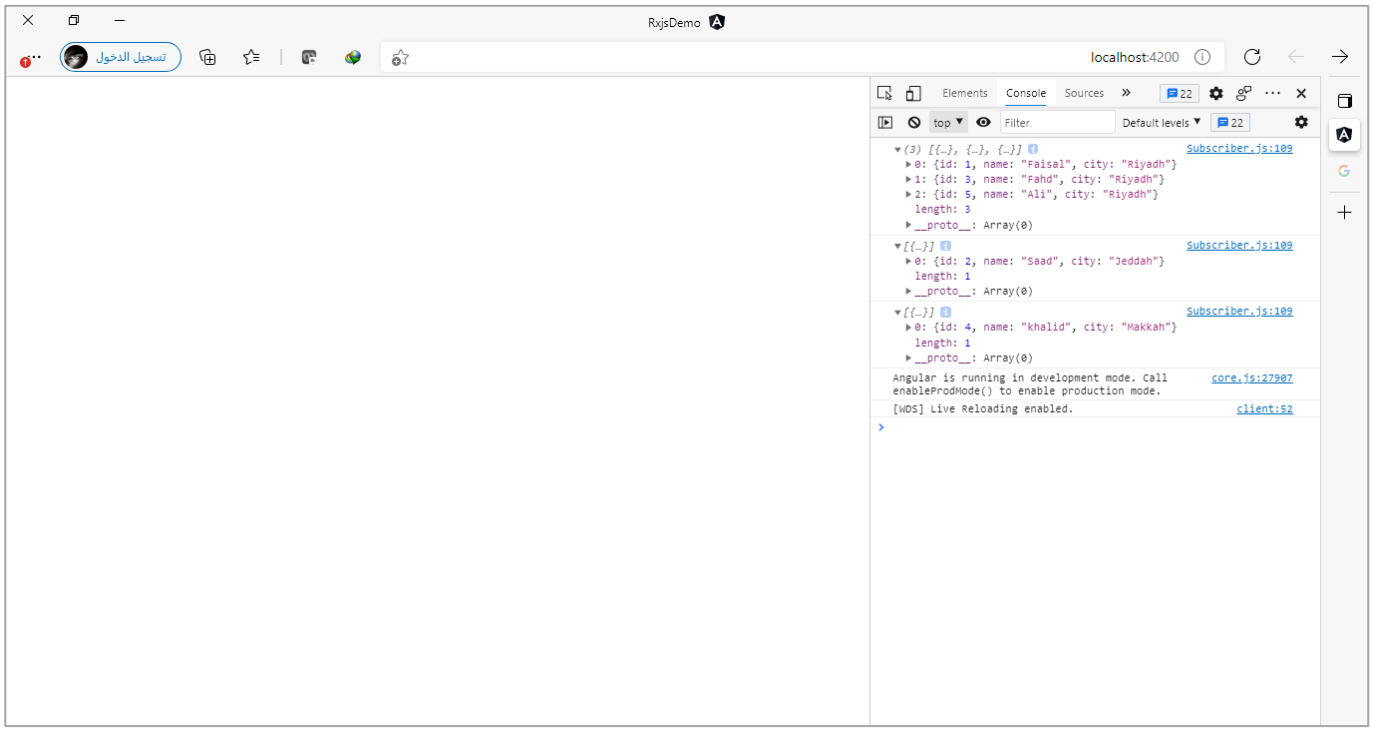
تقوم هذه الدالة بكل بساطة بتجميع مجموعة من البيانات وفق معيار معين نمرره لهذه الدالة على شكل دالة Function، وتستقبل هذه الدالة قيمة وتُعيد Observable لذلك لقراءة هذا الObservable لابد ان نستخدم أحدى دوال Higher Order Mapping، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, Observable } from 'rxjs';
import { groupBy, mergeMap, toArray } from 'rxjs/operators';

export interface IData {
  id: number;
  name: string;
  city: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  data: Observable<IData> = from([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}

  ngOnInit(): void {
    this.data
      .pipe(
        groupBy((value: IData) => value.city),
        mergeMap((valuesGrouped: Observable<IData>) => {
          return valuesGrouped.pipe(toArray());
        })
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم يوجد مجموعة من البيانات ممثلة بالمتغير data، وقمنا بتجميع هذه البيانات بناءً على المدينة بحيث جميع المدن المتشابهة يتم عمل لها Grouping مع بعض ويعمل لها emit كل مجموعة على حدا، ومن ثم استخدمنا

الدالة mergeMap لكي نستقبل الناتج من هذه الدالة والذي هو ايضاً Observable وقمنا فقط بتجميع كل مجموعة على شكل مصفوفة، والناتج سيكون ثلاث مصفوفات الأولى تحتوي على البيانات المشتركة باسم مدينة الرياض، والمصفوفة الثانية تحتوي على البيانات المشتركة باسم مدينة جدة والمصفوفة الأخير باسم مدينة مكة، كالتالي:



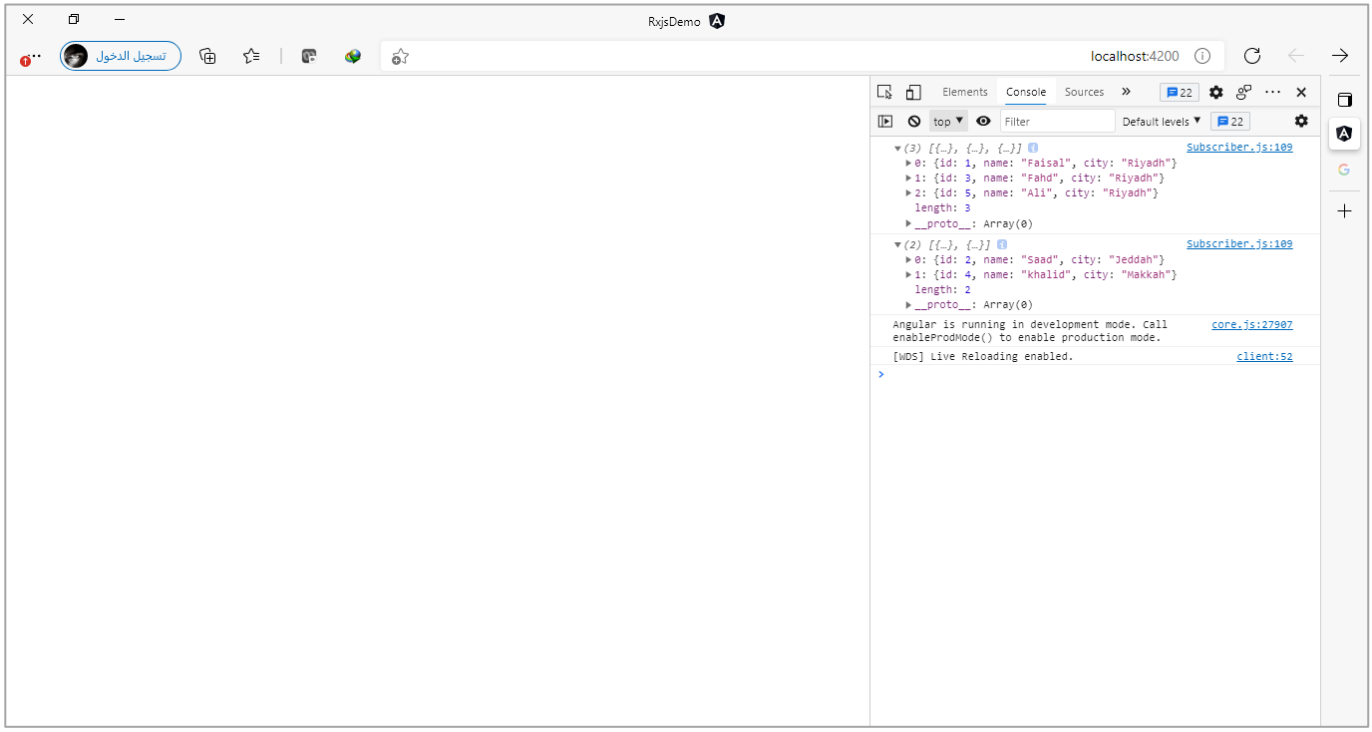
كما ايضاً نستطيع مثلاً ان نقسم النتيجة النهائية إلى مصفوفتين فقط، المصفوفة الأولى تحتوي على جميع البيانات المشتركة باسم مدينة الرياض والمصفوفة الثانية تحتوي على جميع البيانات الأخرى، ويتم هذا الأمر باستبدال الأمر التالي:

```
ngOnInit(): void {
  this.data
    .pipe(
      groupBy((value: IData) => value.city),
      mergeMap((valuesGrouped: Observable<IData>) => {
        return valuesGrouped.pipe(toArray());
      })
    )
    .subscribe(console.log);
}
```

بهذا الأمر:

```
ngOnInit(): void {
  this.data
    .pipe(
      groupBy((value: IData) => value.city.includes('Riyadh')),
      mergeMap((valuesGrouped: Observable<IData>) => {
        return valuesGrouped.pipe(toArray());
      })
    )
    .subscribe(console.log);
}
```

وتكون النتيجة:

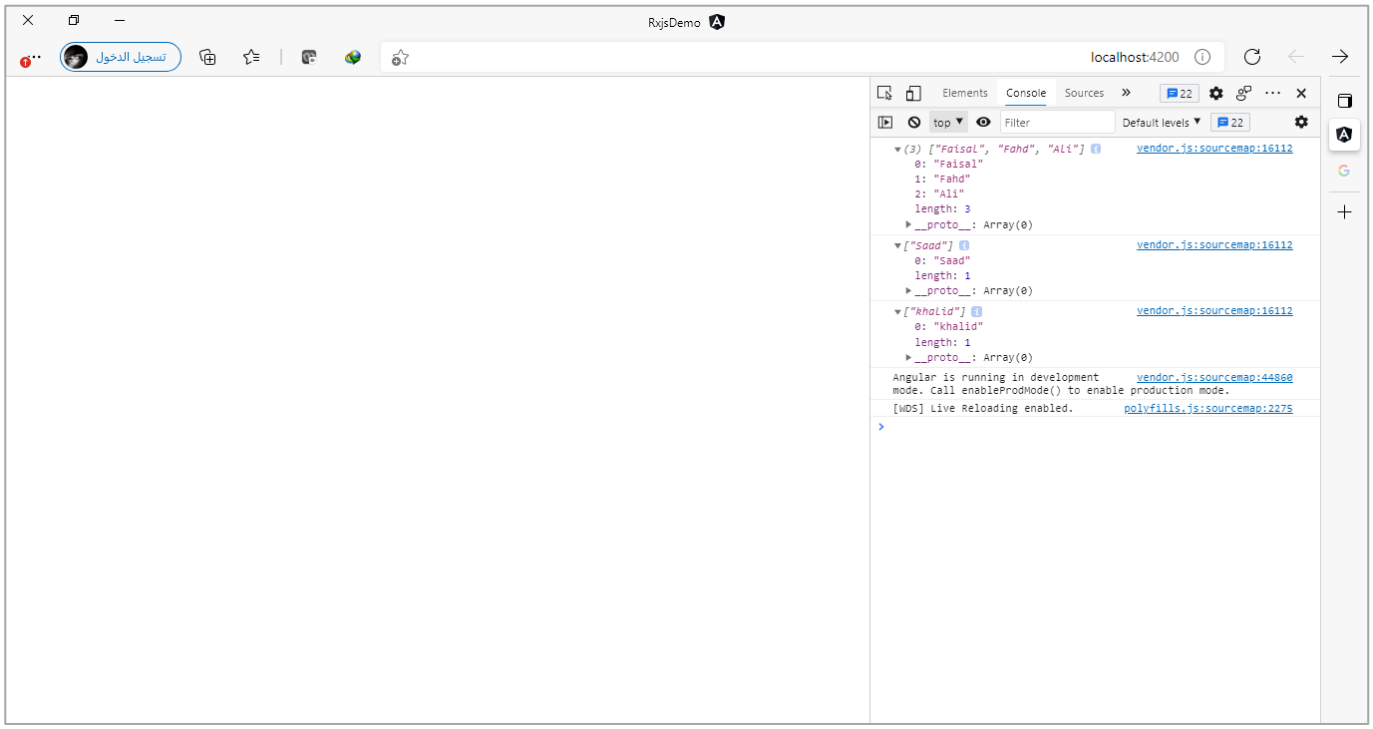


وبما أننا قمنا بتجميع كل بيانات وفق خاصية محددة كما فعلنا في المثال السابق حيث قمنا بتجميع البيانات وفق التشارك باسم المدينة، فمن هذا المنطلق نستطيع ايضاً ان نُعدل في المثال السابق بحيث نقوم بتجميع الأسماء فقط وفق التشارك باسم المدينة، كالتالي:

```
import { Component, OnInit } from '@angular/core';
import { from, Observable } from 'rxjs';
import { groupBy, mergeMap, toArray } from 'rxjs/operators';

export interface IData {
  id: number;
  name: string;
  city: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  data: Observable<IData> = from([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}
  ngOnInit(): void {
```

```
    this.data
      .pipe(
        groupBy(
          (value: IData) => value.city,
          (value: IData) => value.name
        ),
        mergeMap((group: Observable<any>) => group.pipe(toArray()))
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم مررنا لدالة groubBy بارامترين الأول وهو الذي تعرفنا عليه سابقاً واعطينا عليه اكثر من مثال، أما الثاني فهو الذي يمثل التجميعية الفرعية لتجميعية الأساسية، وكأننا نقول لدالة groupBy قومي اولاً بتجميع البيانات بناءً على اسم المدينة ومن ثم في كل تجميعية رئيسية قومي باستخراج الخاصية name فقط من البيانات التي تم تجميعها، وأخيراً قمنا بتجميع هذه البيانات في مصفوفة، وتكون النتيجة كالتالي:



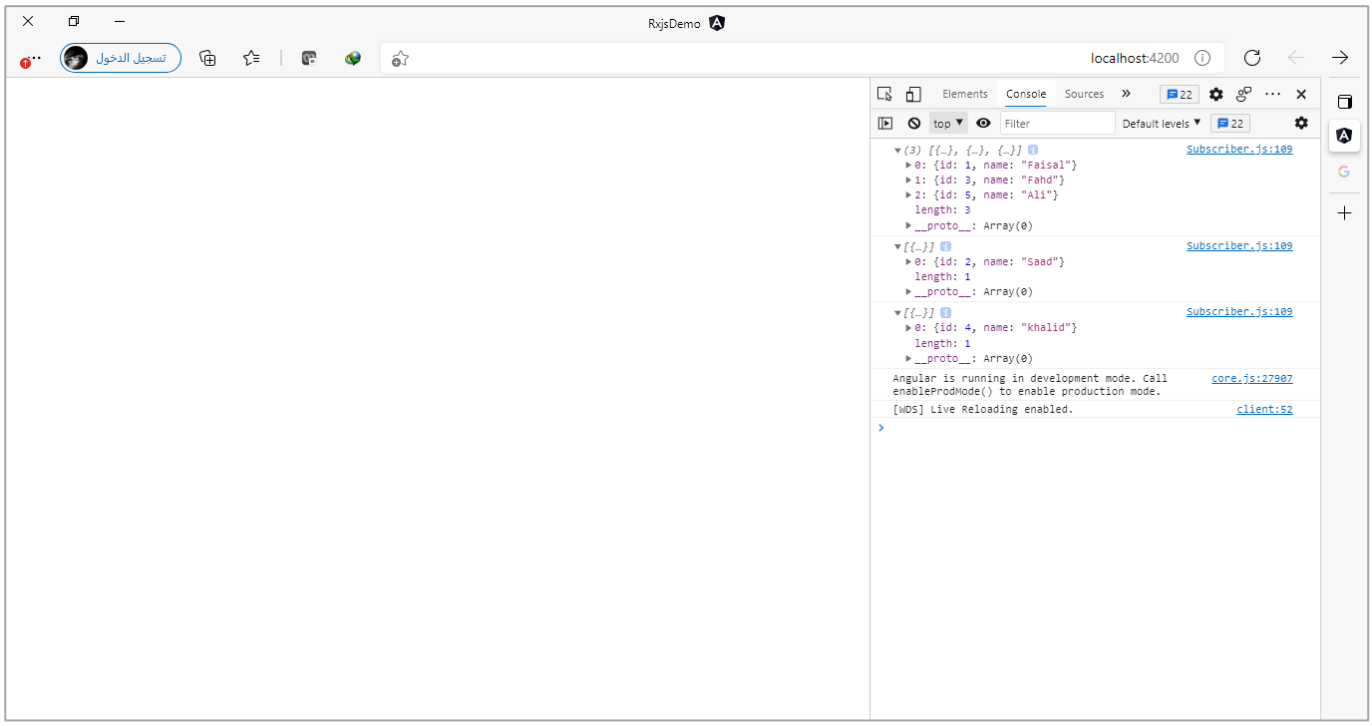
وفي حال أردنا ان نستخرج خاصيتين وهما name وid فهنالك عدة طريق منها وابسطها ان نُعيد هذين الخاصيتين على شكل كائن من خلال استبدال الأمر:

```
this.data
  .pipe(
    groupBy(
      (value: IData) => value.city,
      (value: IData) => value.name
    ),
    mergeMap((group: Observable<any>) => group.pipe(toArray()))
  )
```

بالأمر التالي:

```
this.data
  .pipe(
    groupBy(
      (value: IData) => value.city,
      (value: IData) => {
        const { id, name } = value;
        return { id, name };
      }
    ),
    mergeMap((group: Observable<any>) => group.pipe(toArray()))
  )
  .subscribe(console.log);
```

وتكون النتيجة، كالتالي:



ولكن ماذا لو افترضنا اننا نُريد أن نُحافظ على المعيار الأساسي الذي قمنا بعمل تجميع على أساسه والذي في مثالنا هذا هو city، ويُطلق عليه ايضاً مسمى آخر وهو key، بحيث ندمج هذا key مع البيانات المشتركة معه في النتيجة النهائية.

ونستطيع القيام بهذا الأمر بكل بساطة، حيث ينتج لدينا من الدالة groubBy اثنين Observable، ولقد تعلمنا سابقاً اننا نستطيع دمج اكثر من Observable في Observable واحد باستخدام مجموعة من الدوال الجاهزة مثل zip وconcat وmerge و ...الخ.

والذي يُناسبنا من هذه الدوال هو zip لأننا نُريد دمج كل key في الObservable الأول مع البيانات المشتركة معه والتي تُقابله في الObservable الثاني بعد تحويلها إلى مصفوفة، كالتالي:

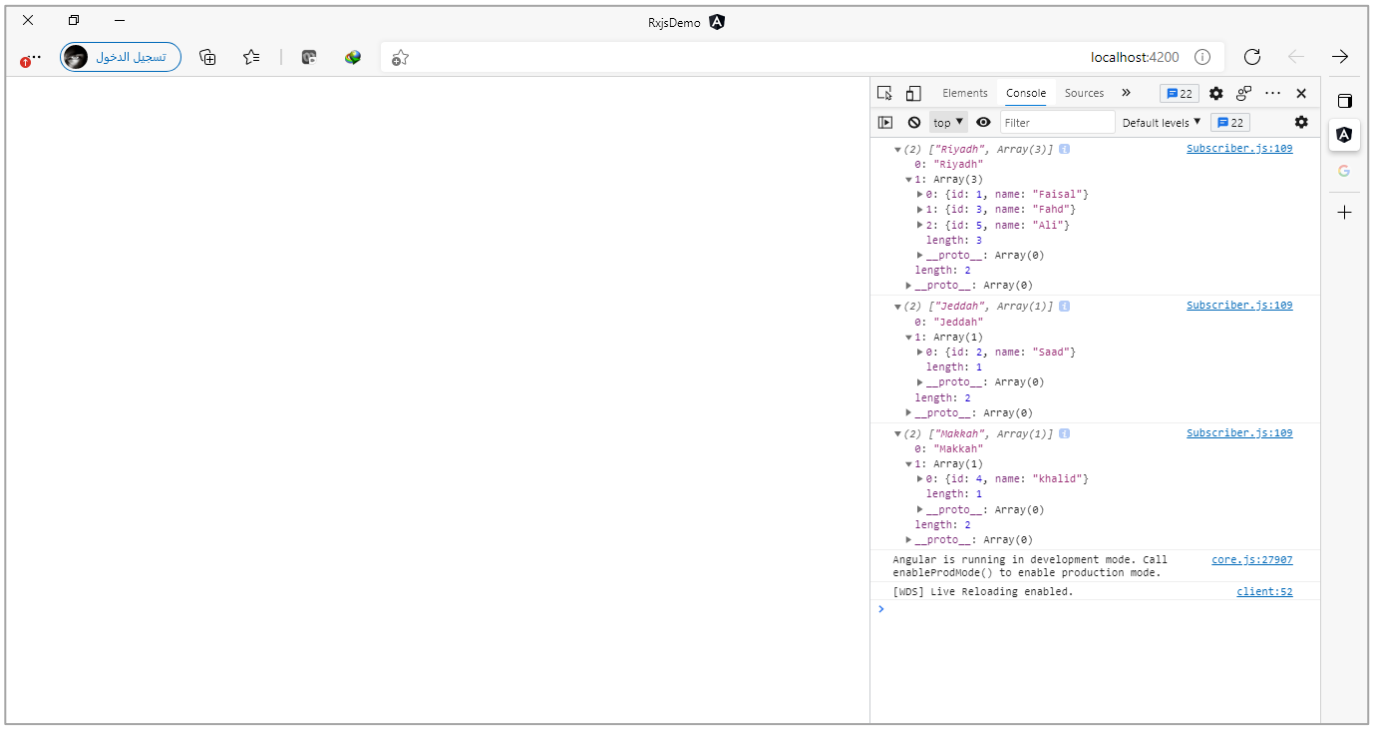
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
```

```
import { from, GroupedObservable, Observable, of, zip } from 'rxjs';
import { groupBy, mergeMap, toArray } from 'rxjs/operators';

export interface IData {
  id: number;
  name: string;
  city?: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  data: Observable<IData> = from([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}
  ngOnInit(): void {
    this.data
      .pipe(
        groupBy(
          (value: IData) => value.city,
          (value: IData) => {
            const { id, name } = value;
            return { id, name };
          }
        ),
        mergeMap((valuesGrouped: GroupedObservable<string, IData>) => {
          return zip(of(valuesGrouped.key), valuesGrouped.pipe(toArray()));
        })
      )
      .subscribe(console.log);
  }
}
```

نلاحظ قمنا بعمل دمج الkey (بعد تحويله إلى Observable لأن key يكون نص لذلك لكي نقوم بدمجه مع الObservable الثاني باستخدام إحدى دوال الدمج وبالتحديد الدالة concat لابد من تحويله إلى Observable) مع الObservable الثاني بعد تحويل قيم الObservable الثاني إلى مصفوفة.

والنتيجة النهائية عبارة عن مجموعة من المصفوفات المصفوفة الأولى تحتوي الkey الأول ولنفرض ان قيمته Riyadh وتحتوي ايضاً على مصفوفة فرعية بداخلها قيم البيانات الأخرى المشتركة جميعها بنفس هذا الkey، وهكذا باقي المصفوفات، ولنشاهد النتيجة في المتصفح، كالتالي:



Filtering Operators -2-3:

وبعدما استعرضنا المجموعة الأولى من دوال pipe والتي تُسمى Transformation Operators بحيث تشترك جميعها في جزء محدد وهو الاتاحة لتعديل وتحويل البيانات من شكل إلى شكل آخر سواء باقتطاع جزء من البيانات او دمج وتحوير البيانات او ...الخ وغيرها من الدوال التي تكلمنا عنها سابقاً، سننتقل الآن إلى مجموعة اخرى من دوال pipe والتي تُسمى Filtering Operators وهي الدوال التي تُتيح لنا كمبرمجين في ترشيح وفلترة او اقتطاع جزء من البيانات، كأن نأخذ او قيمة او آخر قيمة من الObservable او ان نقتطع جزء معين من القيم وفق شرط محدد او ان نبحث عن قيمة محددة، وغيرها الكثير والتي سنتكلم عنها بإذن الله في هذا الجزء.

filter() Operator -1-2-3:

ولعل هذه الدالة من اهم وأشهر هذه المجموعة استخداماً، ليس هذا فحسب نستطيع ايضاً ان نقول انها من الدوال المشهورة وكثيرة الاستخدام في مكتبة Rxjs جنباً إلى جنب مع دوال Mapping.

اما وظيفتها فهي مشابهه لدالة filter في مصفوفات الجافا سكريبت حيث تقوم بعمل فلترة للبيانات وفق شرط معين يُمرر لها على شكل دالة Function، وهذا الشرط يُعيد قيمة منطقية true/false فإذا كانت قيمة الشرط true يسمح بمرور البيانات اما إذا كانت false فسوف يتجاهل هذه البيانات، ومن ثم ينتقل إلى البيانات التالية وايضاً يطبق عليها الشرط المُعطى له، وهكذا إلى ان ينتهي من جميع البيانات.

وهنالك نقطة مهمة في هذه الدالة انها لا تتعامل مع البيانات الObservable التي تكون على شكل مصفوفة بشكل مباشر، وانما يجب ان يتم عملها لها flatting أي تجزئة لكي تستطيع قراءة كل بيانات لوحدها وتطبيق الشرط عليها.

ولتوضيح، لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, Observable } from 'rxjs';

export interface IData {
  id: number;
  name: string;
  city?: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  data: Observable<IData> = from([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}
  ngOnInit(): void {
  }
}
```

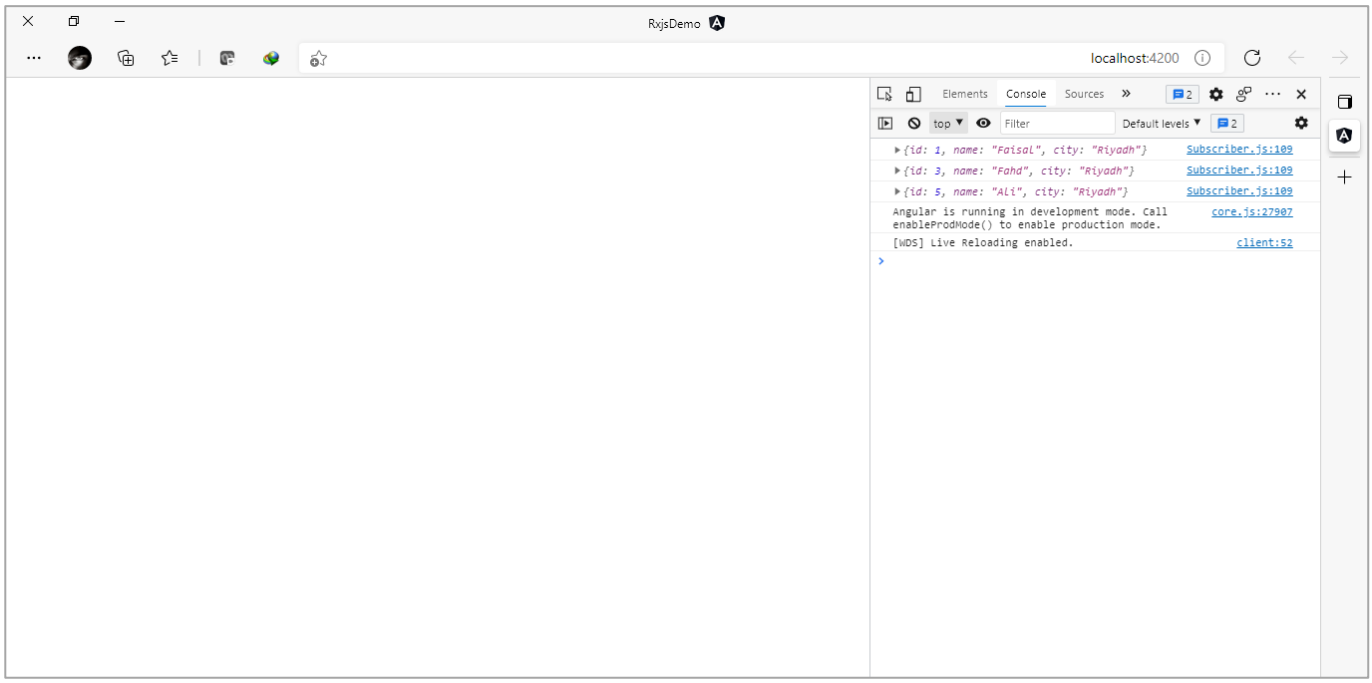
كما نلاحظ في الأعلى لدينا مجموعة من البيانات وهذه البيانات قمنا بقراءتها باستخدام الدالة from وكما هو معروف ان هذه الدالة تقوم بعمل flatten للبيانات أي تجزئة هذه البيانات، الآن نُريد مثلاً ان نقوم بعمل فلترة لهذه البيانات بحيث نستقطع منها الأشخاص الذين يسكنون في Riyadh، وللقيام بهذا الامر سوف نستخدم الدالة filter، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, Observable } from 'rxjs';
import { filter } from 'rxjs/operators';

export interface IData {
  id: number;
  name: string;
  city?: string;
}
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  data: Observable<IData> = from([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}
  ngOnInit(): void {
    this.data
      .pipe(
        filter((val: IData) => val.city === 'Riyadh'),
      )
      .subscribe(console.log);
  }
}
```

كما هو واضح استخدمنا الدالة filter ومررنا لها دالة function بدون اسم وهذه الfunction تستقبل بارامتر اسميته val بحيث يتم عن طريقه تخزين البيانات والقيم القادمة لنا من الObserver، اما من ناحية محتوى هذه الfunction فهو بكل بساطة شرط بسيط وهو هل القيم القادمة لنا من هذا الObserver والمتمثل في البارامتر val يساوي Riyadh فإذا كان يساوي سيتم إعادة قيمة منطقية true وسيتم السماح لهذه البيانات بالمرور إلى دوال Pipes الأخرى في حال وجودها او إلى Subscription مباشرة، ومن ثم ينتقل إلى البيانات الثانية وايضاً سيتم تخزينها في val مع حذف البيانات السابقة، وايضاً سيطبق نفس الشرط لكي يتم السماح او تجاهل هذه البيانات، وفي النهاية ستكون النتيجة كالتالي:



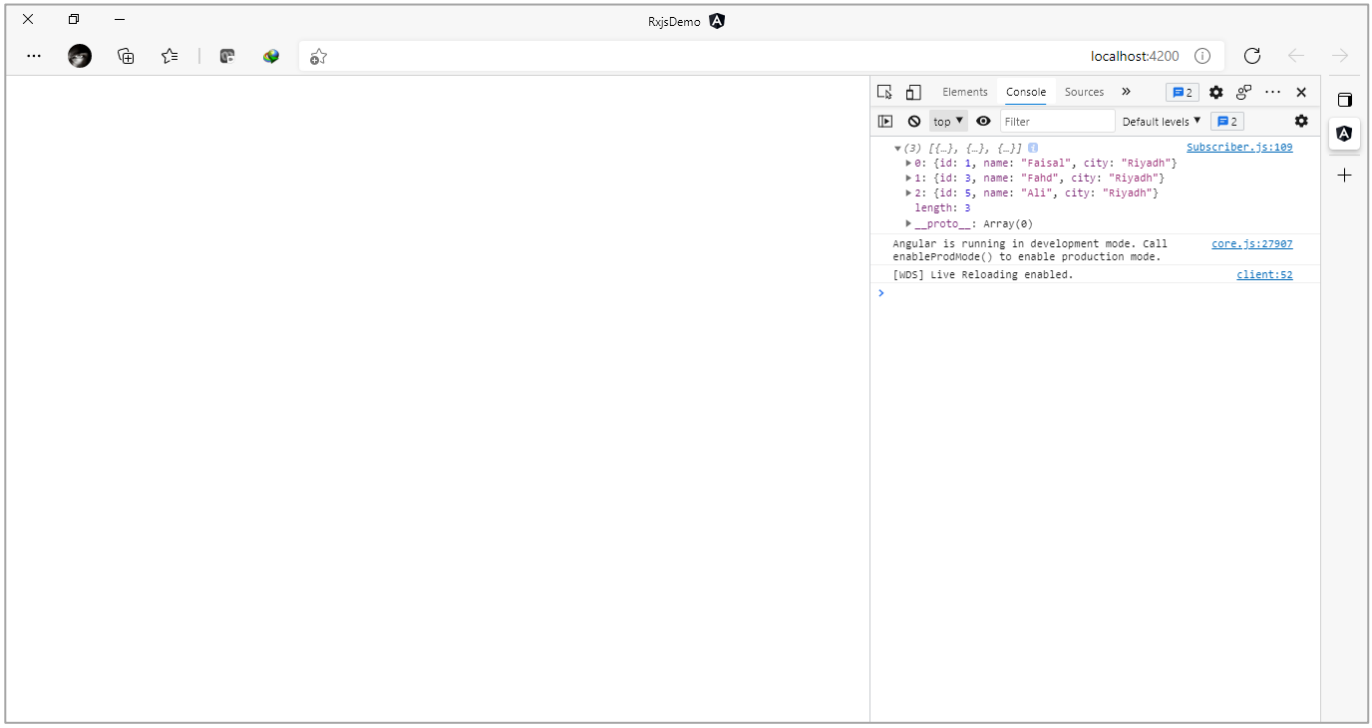
كما نلاحظ تم عرض البيانات مُجزئة وفي حال اردنا ارجاعها إلى مصفوفة نستخدم الدالة toArray بعد الدالة filter.

اما ألان لنفرض ان البيانات قادمة لنا من الObserver على شكل مصفوفة، ففي هذه الحالة لدينا مجموعة من الحلول منها استخدام الدالة map وفي داخل الدالة map نستخدم الدالة filter الخاصة بالمصفوفات وليست الدالة الخاصة بمكتبة Rxjs، والحل الثاني هو ان نعمل flatting للبيانات باستخدام إحدى دوال Higher Order Mapping ومن ثم نستخدم الدالة filter ومن ثم نستخدم الدالة toArray لكي نُرجعها لشكلها الأساسي وهي مصفوفة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from, Observable, of } from 'rxjs';
import { filter, mergeAll, toArray } from 'rxjs/operators';

export interface IData {
  id: number;
  name: string;
  city?: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  data: Observable<IData[]> = of([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}
  ngOnInit(): void {
    this.data
      .pipe(
        mergeAll(),
        filter((val: IData) => val.city === 'Riyadh'),
        toArray()
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم استخدمنا الدالة of لكي نجعل قيم الObservable على شكل مصفوفة، ومن ثم استخدمنا الدالة mergeAll لكي نُجزئ هذه المصفوفة ومن ثم مررنا هذه البيانات على الدالة filter واخيراً ارجعنا البيانات إلى شكلها الطبيعي على شكل مصفوفة، وسوف تكون النتيجة بشكلها النهائي كالتالي:



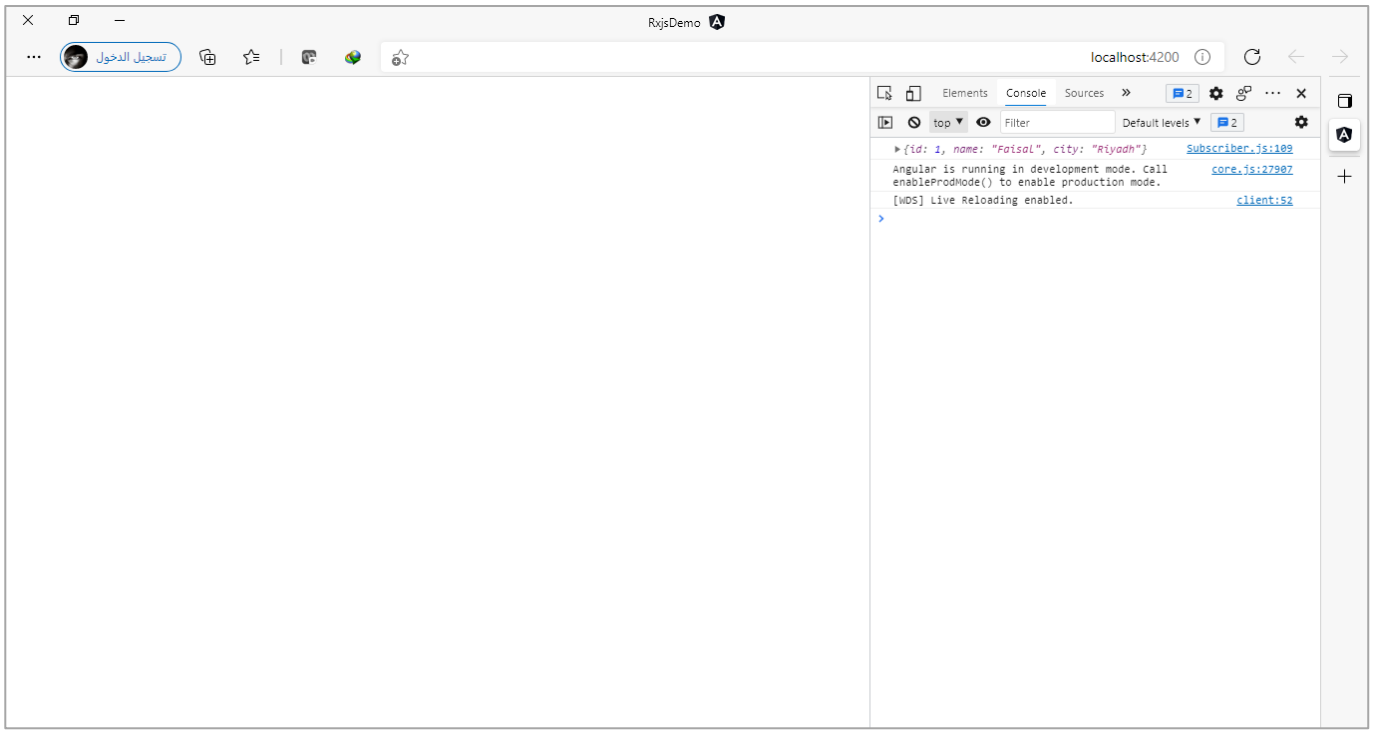
:find() Operator -2-2-3

هذه الدالة مشابهه لدالة filter ولا تختلف عنها إلا أنها تقوم بقراءة او بيانات ينطبق عليها الشرط وتتجاهل الباقي، لذلك نستخدمها في حال أردنا البحث عن بيانات وحيدة، كأن نبحث عن رقم موظف وحيد – لا يتكرر – من بين مجموعة من الأرقام، وهذه الدالة بذلك مشابهه لدالة find في مصفوفات الجافا سكريبت، ولتوضيح لنعطي المثال التالي:

```
import { Component, OnInit } from '@angular/core';
import { Observable, of } from 'rxjs';
import {  find, mergeAll } from 'rxjs/operators';

export interface IData {
  id: number;
  name: string;
  city?: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  data: Observable<IData[]> = of([
    { id: 1, name: 'Faisal', city: 'Riyadh' },
    { id: 2, name: 'Saad', city: 'Jeddah' },
    { id: 3, name: 'Fahd', city: 'Riyadh' },
    { id: 4, name: 'khalid', city: 'Makkah' },
    { id: 5, name: 'Ali', city: 'Riyadh' },
  ]);
  constructor() {}
  ngOnInit(): void {
```

```
    this.data
      .pipe(
        mergeAll(),
        find((val: IData) => val.city === 'Riyadh')
      )
      .subscribe(console.log);
  }
}
```

وبما انه وحيد قمت بحذف دالة toArray ولكن في حال أردت عزيزي المتعلم في تطبيقك لأي سبب من الأسباب ان تكون النتيجة النهائية على شكل مصفوفة فلا ضير من استخدام الدالة toArray.



3-2-3- Taking Operators:

والمقصود فيها الدوال التالية:

- take()
- takeUntil()
- takeWhile()
- takeLast()

وهذه الدوال بصفة عامة تقوم باقتطاع جزء من البيانات ومن ثم تقوم بإغلاق الObservable، ولكنها تختلف في كيفية هذا الاقتطاع وآلية إيقاف الObservable.

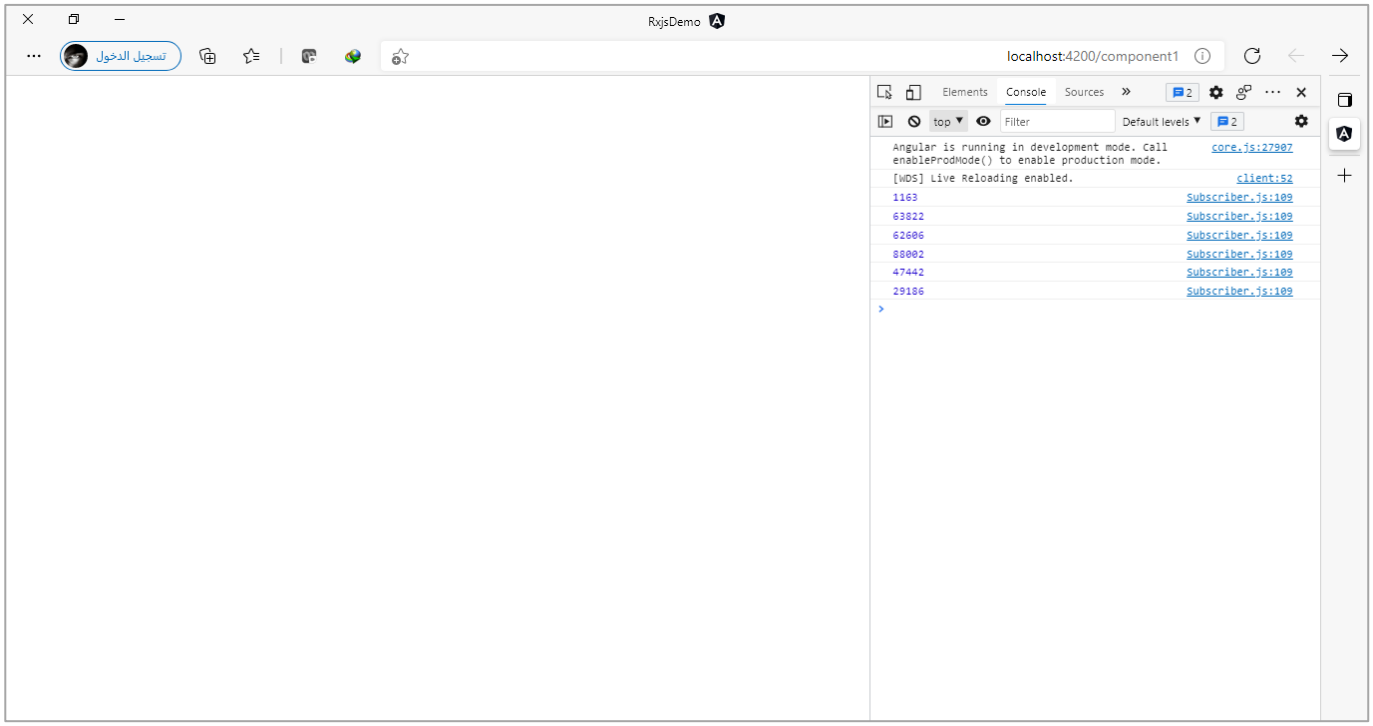
take() -1-3-2-3:

هذه الدالة تقوم باقتطاع جزء من البيانات معلوم لدينا حيث تستقبل قيمة رقمية وهذه القيمة تُعبر عن عدد البيانات التي نُريد اقتطاعها من الObservable، ومن ثم يتم اغلاق هذا الObservable، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { map, take } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    interval(1000)
      .pipe(
        map((value: number) => Math.floor(Math.random() * 100000) + value),
        take(6),
      )
      .subscribe(console.log);
  }
}
```

كما هو واضح قمنا باستخدام دالة interval لكي نولد Observable من القيم الرقمية و، بحيث كُل ثانية يتم عمل emit لقيمة رقمية وهذه القيمة قمنا بالتلاعب فيها وتحويلها إلى قيمة عشوائية باستخدام الدالة map، ومن ثم استخدمنا الدالة take لأخذ اول 6 قيم فقط ومن ثم نغلق هذا الObservable.



takeUntil() -2-3-2-3:

اما هذه الدالة فنستخدمها في حالة ان القيم غير معروفة لدينا، وانما في حال حدوث شيء ما نقوم بإغلاق هذا الObservable، وهذا الشيء هو Observable آخر، حيث هذه الدالة تستقبل Observable وعندما يقوم هذا الObservable بعمل emit لقيمه ستقوم هذه الدالة تلقائياً بعمل اغلاق للObservable الأول والاكتفاء بالقيم التي تم عمل لها emit من قبل.

مع العلم ان هذه الدالة تعتبر من الدوال المشهورة وكثيرة الاستخدام من قبل المطورين، ومن اشهر استخداماتها في Angular في حال كان لدينا component يقوم بعمل subscribe لObservable من خارج هذا الcomponent كأن يكون مثلاً في service، ونُريد عند الانتقال من هذا الcomponent إلى component آخر ان يتم اغلاق هذا الObservable لكي لا يُرهق ويستهلك ذاكرة جهاز مستخدم التطبيق الخاص بك، وعند الرجوع إلى هذا الcomponent مرة أخرى نُعيد subscribe وجلب البيانات.

ولتوضيح لنفرض انه لدينا اثنين component بحيث الcomponent الأول هو الذي يقوم بعمل subscription للObservable في ملف service اما الثاني فلا يعمل شيء وانما وضعناه لكي ننتقل من الcomponent الأول إليه ويوجد به زر لكي نرجع مرة أخرى للcomponent الأول.

ومما قيل سابقاً قم عزيزي المتعلم بإنشاء ملف service جديد وعدد اثنين components، وفي حال كنت لا تعرف انصحك بمراجعة كتابي angular components and services وهو جزء من هذه السلسة، وتم فيه شرح كل شيء تريده.

وبعد الانتهاء من انشاء الملفات، سنذهب اولاً إلى ملف service، ونضيف السطر التالي:

ملف test.service.ts

```
import { Injectable } from '@angular/core';
import { interval, Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class TestService {
  public timer: Observable<number> = interval(1000);
  constructor() {}
}
```

وفي ملف app-routing.module.ts، نقوم بضبط إعدادات وتهيئة Routing، وايضاً عزيزي المتعلم في حال عدم معرفتك بكيفية التعامل مع الRouting انصحك بالاطلاع على كتابي angular routing and modules وهو جزء من هذه السلسلة

ملف app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { SubComponent } from './sub/sub.component';
```

```
import { Sub2Component } from './sub2/sub2.component';

const routes: Routes = [
  { path: '', component: AppComponent },
  { path: 'component1', component: SubComponent },
  { path: 'component2', component: Sub2Component },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

ومن ثم نذهب إلى ملف app.component.html ونُضيف الMarkup التالي:

ملف app.component.html

```
<h1>Home Component</h1>
<hr />
<router-outlet></router-outlet>
```

اما الآن لنذهب إلى الcomponent الأول والذي اسميته sub وبالتحديد إلى ملف الclass لكي نستدعي الObservable من ملف service ومن ثم نعمل لها subscribe، بدون ان نستخدم الدالة takeUntil، لكي نُشاهد المشكلة التي حلتها لنا هذه الدالة.

ملف sub.component.ts

```
import { Component, OnInit } from '@angular/core';
import { TestService } from '../test.service';

@Component({
  selector: 'app-sub',
  templateUrl: './sub.component.html',
  styleUrls: ['./sub.component.scss'],
})
export class SubComponent implements OnInit {
  constructor(private test: TestService) {}
  ngOnInit(): void {
    this.test.timer.subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم استدعينا الخاصية timer والتي تمثل الObservable من ملف service ومن ثم عملنا لها subscribe، والآن لنذهب إلى ملف template لنفس الcomponent لكي نُضيف زر التوجه إلى component الثاني.

ملف sub.component.html

```
<div style="display: flex; flex-direction: column; align-items: center">
  <h1 class="m-5">Component1</h1>
```

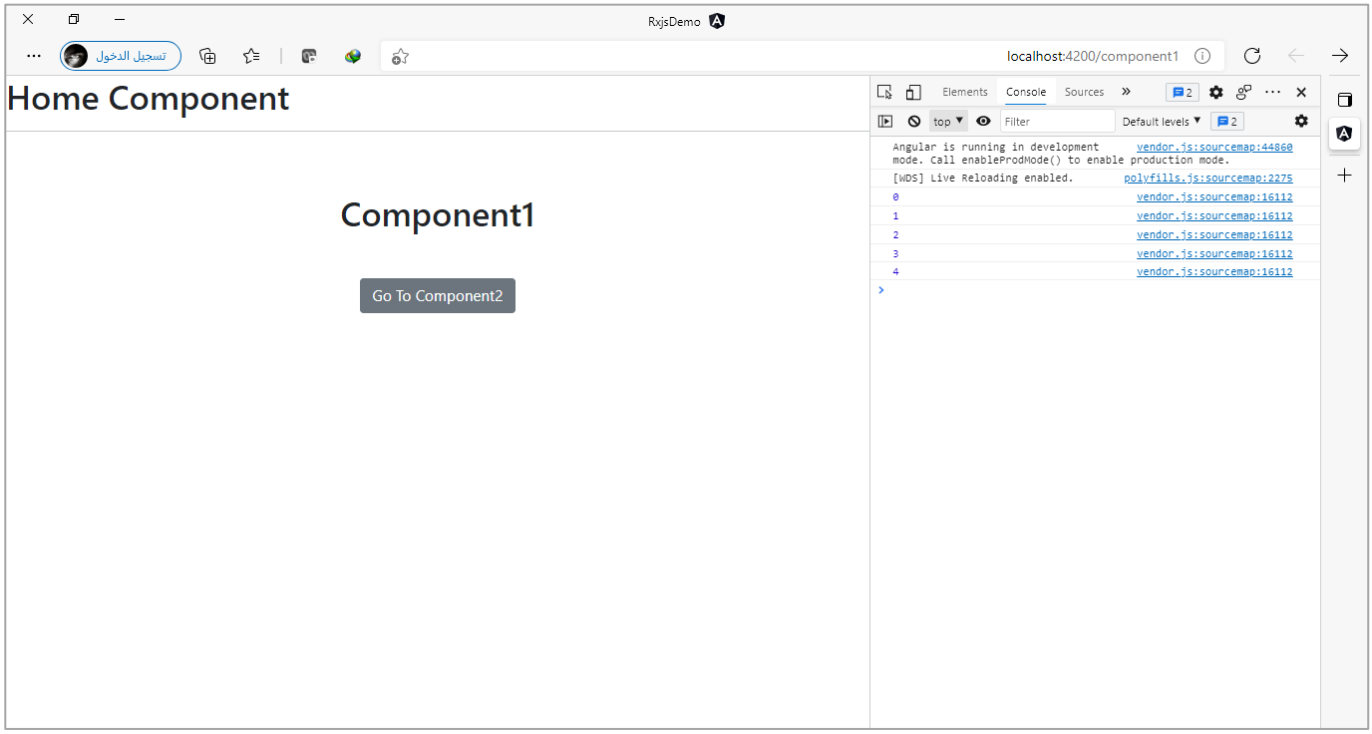
```
  <a class="btn btn-secondary" routerLink="/component2">Go To Component2</a>
</div>
```

اما component الثاني فكُل الذي سوف نُضيفه هو زر للرجوع إلى الcomponent الأول، كالتالي:

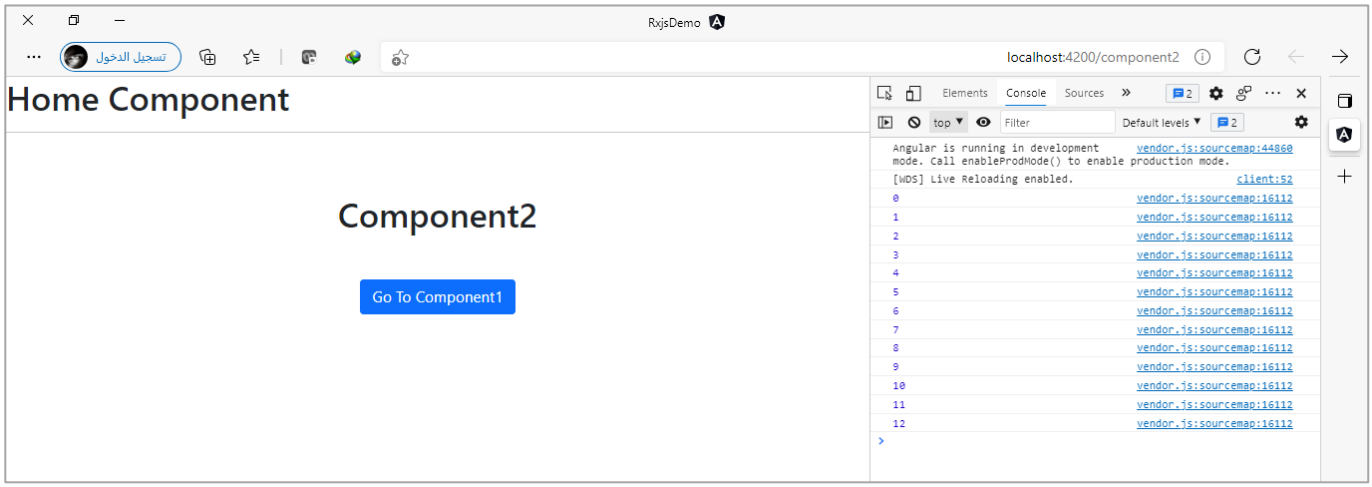
ملف sub2.component.html

```
<div style="display: flex; flex-direction: column; align-items: center">
  <h1 class="m-5">Component2</h1>
  <a class="btn btn-primary" routerLink="/component1">Go To Component1</a>
</div>
```

اما الآن لنشاهد النتيجة في المتصفح، ومن ثم نعلق على المشكلة وكيفية حلها:



نلاحظ تم تشغيل component1 وبنفس الوقت تم عمل subscription وعرض القيم في الconsole من الObservable القادم من الservice، والآن لنقم بالضغط على زر Go To Component2 لكي ننتقل اليه وهذا الإجراء سينقلنا إلى component2 وتدمير الcomponent1 بما فيه من اكواد وشفرات برمجية، كالتالي:



نلاحظ تم الانتقال إلى component2 ولكن لايزال subscription يستقبل ويعرض البيانات، على الرغم من انه تم تدمير component1، وعدم حاجتنا لهذه القيم، وهذا يؤدي إلى استهلاك موارد الجهاز لدى مستخدمي تطبيقك، لذلك نحتاج إن نغلق هذا الObservable في حال عدم الحاجة إليه، وهنالك عدة طرق تقدمها لنا مكتبة Rxjs منها استخدام دالة takeUntil حيث نستطيع مثلاً عند تدمير component1 وبالتحديد في دالة lifecycle hook ذات الاسم Destroy ننفذ أمر معين وهذا الامر عبارة عن Observable ويتم تمريره إلى الدالة takeUntil بحيث اول ما يتم عمل emit لأي قيمة – هنا لا نهتم بالقيم وانما فقط ما يهمنا ان تكون قيمة على شكل Observable – سيتم تفعيل الدالة takeUntil وإيقاف هذا الObservable، الأساسي، ولنكتفي الآن بالسرد النظري ولننتقل إلى الجزء العملي وكتابة الشفرات البرمجية، حيث سنستخدم الدالة subject ونمررها إلى الدالة takeUntil وفي دالة Destroy نعمل emit عن طريق الدالة subject، وكما فلنا سابقاً لا يهمنا في هذا المثال ماذا تكون هذه القيمة فقط ان هذا الObservable قام بعمل emit لقيمة ولو كانت فارغة، كالتالي:

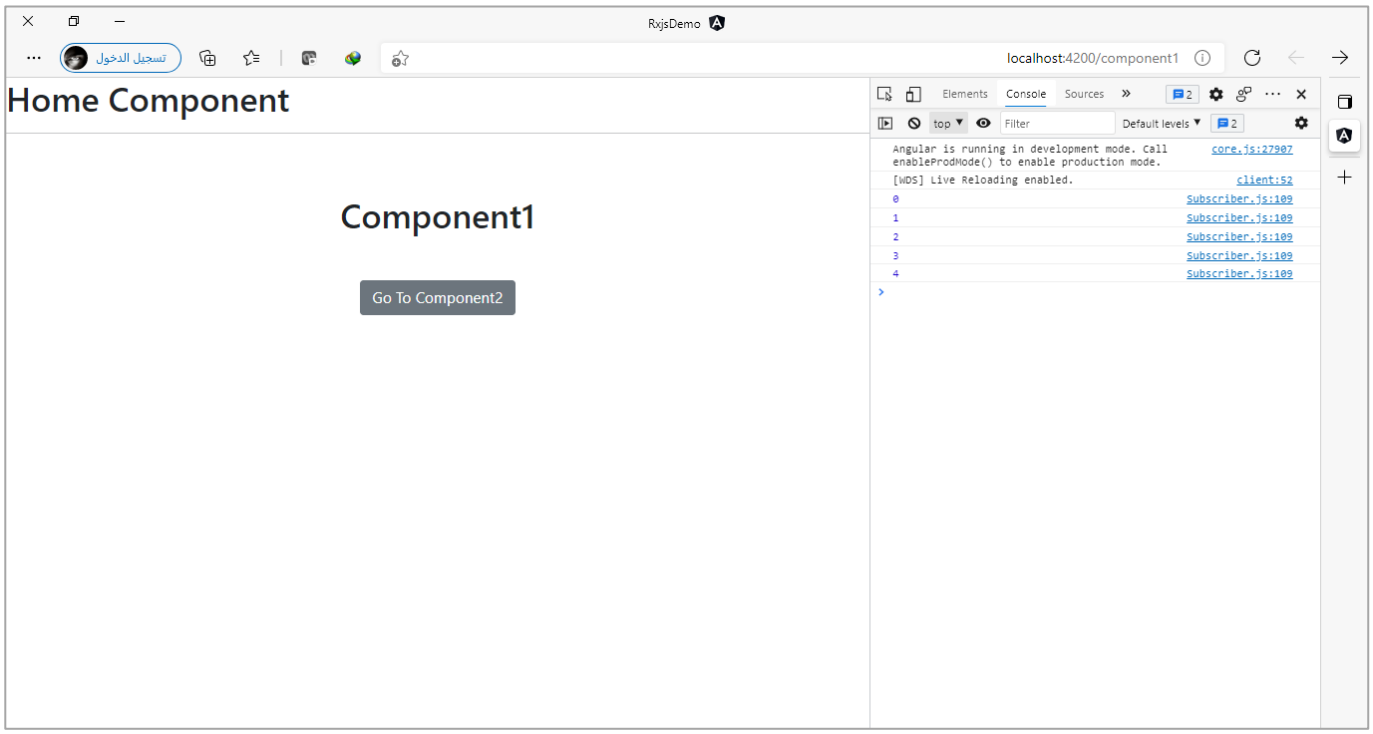
ملاحظة: في حال كان لديك ضبابية وعدم فهم لكيفية التعامل مع components ودوال Lifecycle hooks تستطيع عزيزي المتعلم الرجوع إلى كتابي Angular Components and Services وهو جزء من هذه السلسلة.

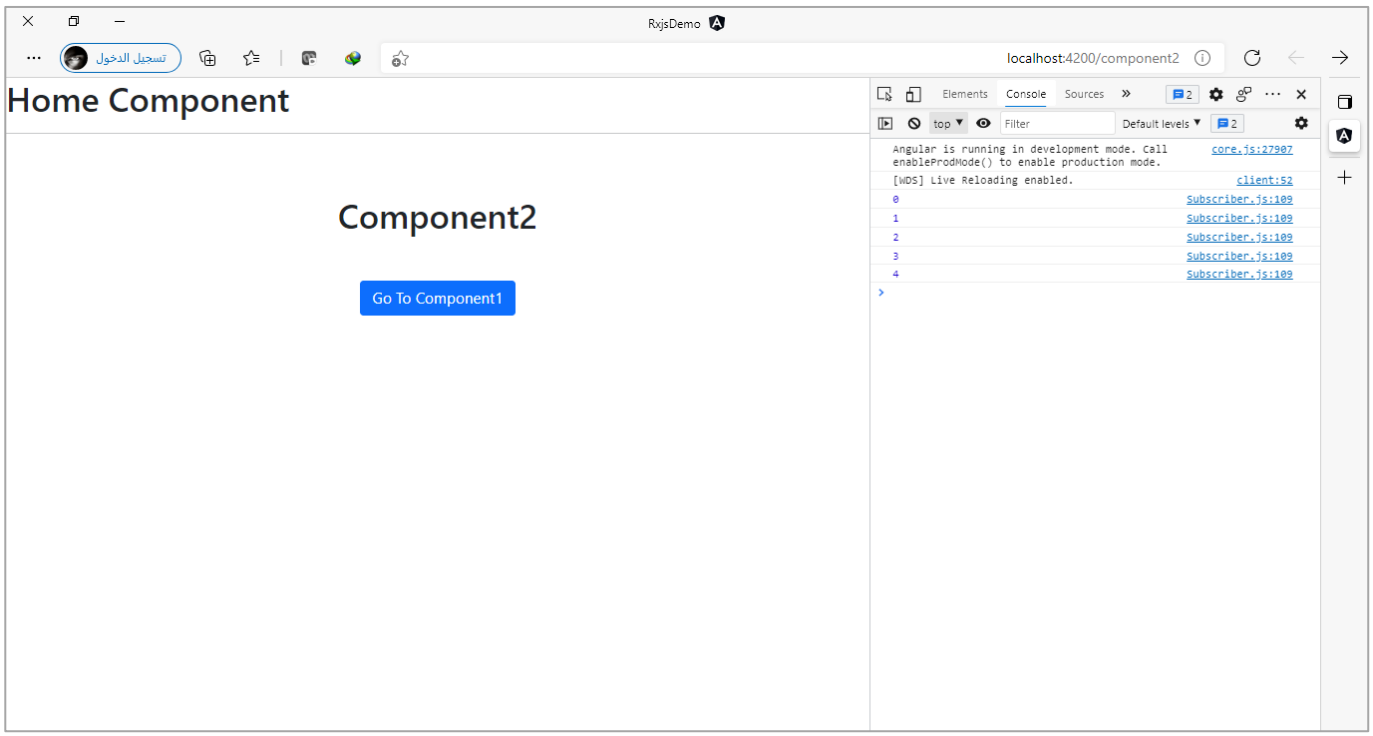
ملف sub.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';
import { TestService } from '../test.service';

@Component({
  selector: 'app-sub',
  templateUrl: './sub.component.html',
  styleUrls: ['./sub.component.scss'],
})

export class SubComponent implements OnInit, OnDestroy {
  private notifier = new Subject();
  constructor(private test: TestService) {}
  ngOnInit(): void {
    this.test.timer.pipe(takeUntil(this.notifier)).subscribe(console.log);
  }

  ngOnDestroy(): void {
    this.notifier.next('');
  }
}
```

كما نلاحظ عرفنا متغير على انه subject واسميته notifier – لك حرية اختيار الاسم الذي تُريده – بحيث يكون الObservable الذي سنمرره إلى الدالة takeUntil بحيث تمثل الشرط الذي عن طريقه يتم إيقاف الObservable الأساسي، واخيراً في دالة ngOnDestroy تم عمل emit لأي قيمه فنحن كما اشرنا بأكثر من مره لا يهمنا في مثالنا هنا

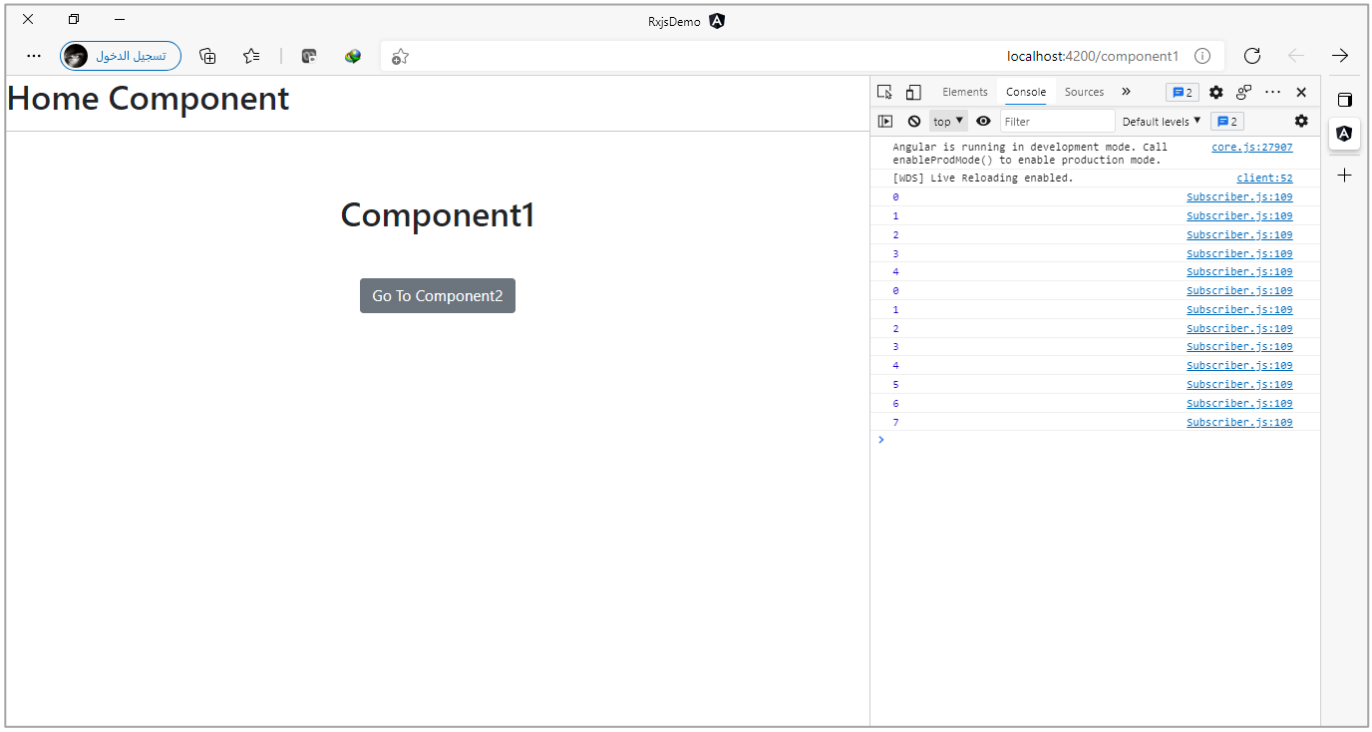
القيم فالذي يهمنا هو ان يتم فقط عمل emit لأي قيمه بواسطة هذا الObservable لكي تلتقط الدالة takeUntil هذا الObservable وتغلق الObservable الأساسي الذي اسميناه timer، اما الآن لنشاهد النتيجة كالتالي:





كما نلاحظ عند الانتقال إلى الcomponent2 تم اغلاق وإيقاف الObservable وهذا هو المتوقع.

وعند الرجوع مرة أخرى إلى component1 يتم إعادة استقبال الObservable من البداية، كالتالي



واخيراً تجدر الإشارة انه ليس الزاماً ان تعمل emit لقيمة فارغة ولكن هنا في مثالنا لا نهتم في القيم وقد تحتاج عزيزي المتعلم في تطبيقك ان تعمل emit لقيم معينة.

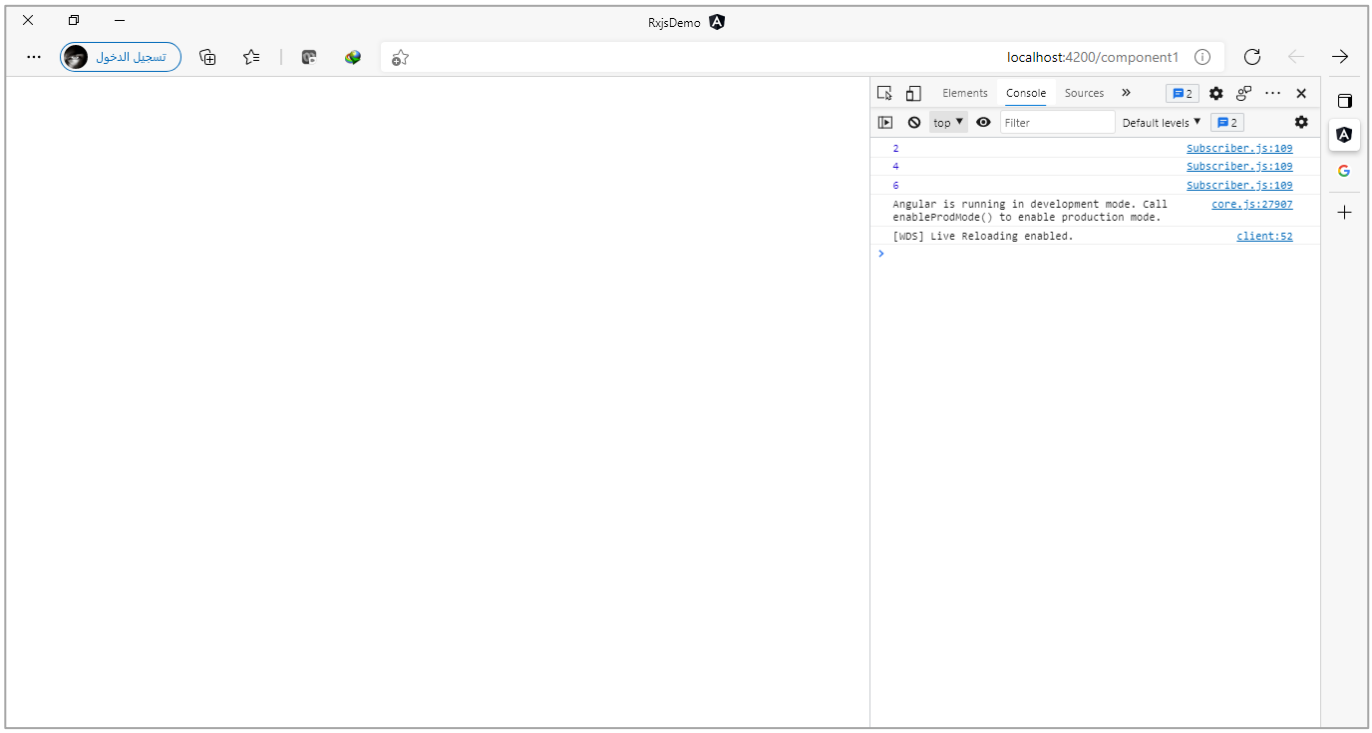
takeWhile() -3-3-2-3:

هذه الدالة بسيطة حيث تقوم بنفس مهام اخواتها سابقات الذكر، والفرق انها هنا تستقبل دالة function وعن طريق هذه الدالة نُحدد متى يتم إيقاف الObservable، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { takeWhile } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 6, 3, 8, 10)
      .pipe(
        takeWhile((val: number) => {
          return val % 2 === 0;
        })
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ عزيزي المتعلم لدينا Observable يحتوي على مجموعة من الأعداد، وقمنا بتمريره على الدالة takeWhile لكي نقتطع جزء معين منه وفق شرط معين ومن ثم بشكل تلقائي تقوم هذه الدالة بإغلاق هذا الObservable وتجاهل باقي القيم، والشرط الذي مررناه لدالة هو قم باقتطاع الإعداد الزوجية فقط، وتكون النتيجة كالتالي:

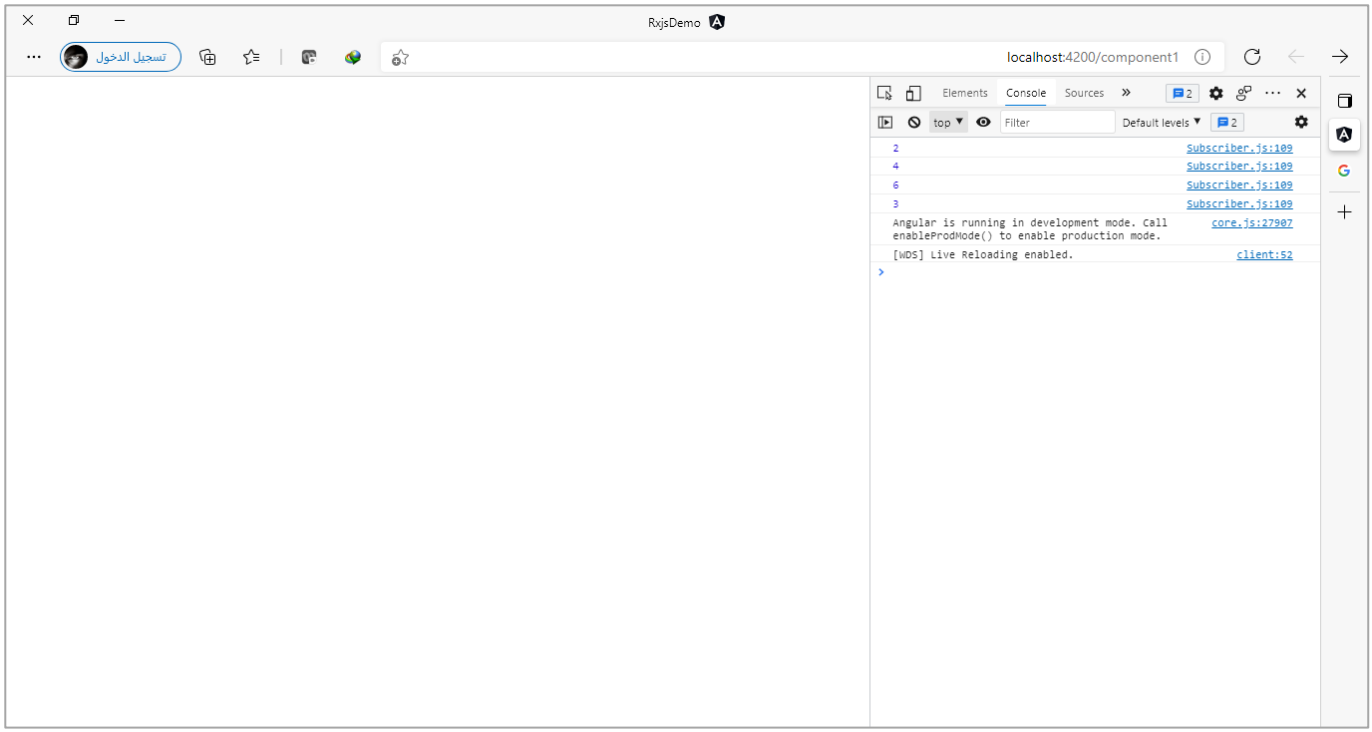


نلاحظ عندما إعادة الدالة false أي لم يتم تحقق الشرط (اقصد بالدالة هي البارامتر التي تم تمريرها لدالة takeWhile على شكل دالة Function) تم تجاهل باقي القيم الأخرى 8 و10 على الرغم من انها اعداد زوجيه، ولكن هذه طبيعة دالة takeWhile عندما لم يتحقق الشرط تكتفي بالقيم الموجودة لديها وقد حققت الشرط وتجاهل الباقي مع اغلاق الObservable، وهي بذلك تختلف عن الدالة filter التي تقرأ جميع قيم الObservable وفي حال تحقق الشرط تقرأ القيمة وتتجاهل القيم التي لم تحقق الشرط.

كما ان الدالة takeWhile تستقبل بارامتر اختياري آخر وتكون قيمته منطقيه true/false وقيمته الافتراضيه false وفي حال قمنا بتغييرها إلى true فإن هذه الدالة تقرأ القيم التي **لم تحقق الشرط**، كالتالي:

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { takeWhile } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 6, 3, 8, 10)
```

```
      .pipe(
        takeWhile((val: number) => {
          return val % 2 === 0;
        }, true)
      )
      .subscribe(console.log);
  }
}
```



كما تلاحظ عزيزي المتعلم تم قراءة القيمة 3 وهي القيمة التي بسببها بم يتحقق الشرط وبالتالي قامت الدالة takeWhile بإغلاق الObservable وتجاهل بقية القيم.

takeLast() -4-3-2-3:

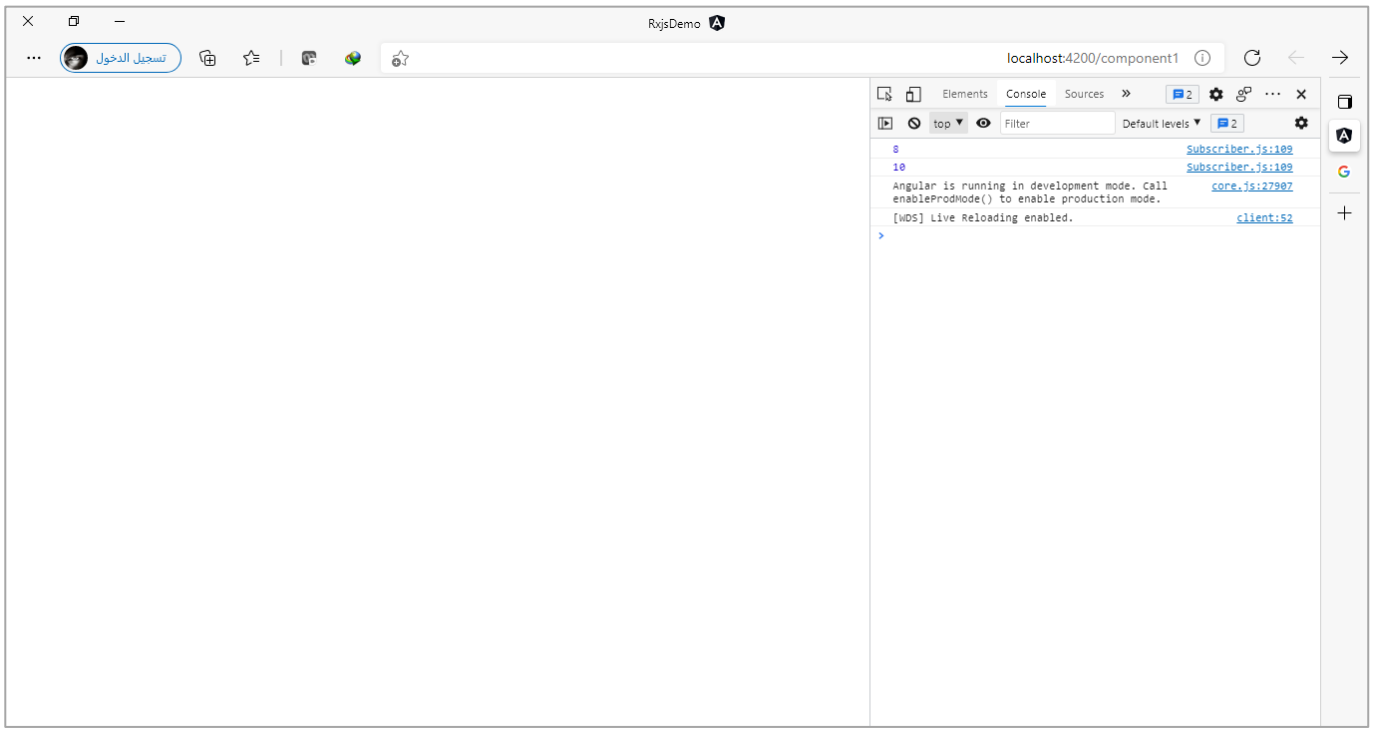
اما هذه الدالة فهي عكس الدالة take حيث تستقطع مجموعة من القيم من الObservable في نهاية القيم وليس في البداية كما هو الحال في الدالة take، كما في المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { takeLast } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {

  constructor() {}
  ngOnInit(): void {
    of(2, 4, 6, 3, 8, 10).pipe(takeLast(2)).subscribe(console.log);
  }
}
```

والنتيجة، ستكون القيمة 8 و10 وهما آخر قيمتين من الObservable لأننا مررنا العدد 2 إلى الدالة takeLast، كالتالي:



3-2-4- Skipping Operators:

والمقصود فيها الدوال التالية:

- skip()
- skipUntil()
- skipWhile()
- skipLast()

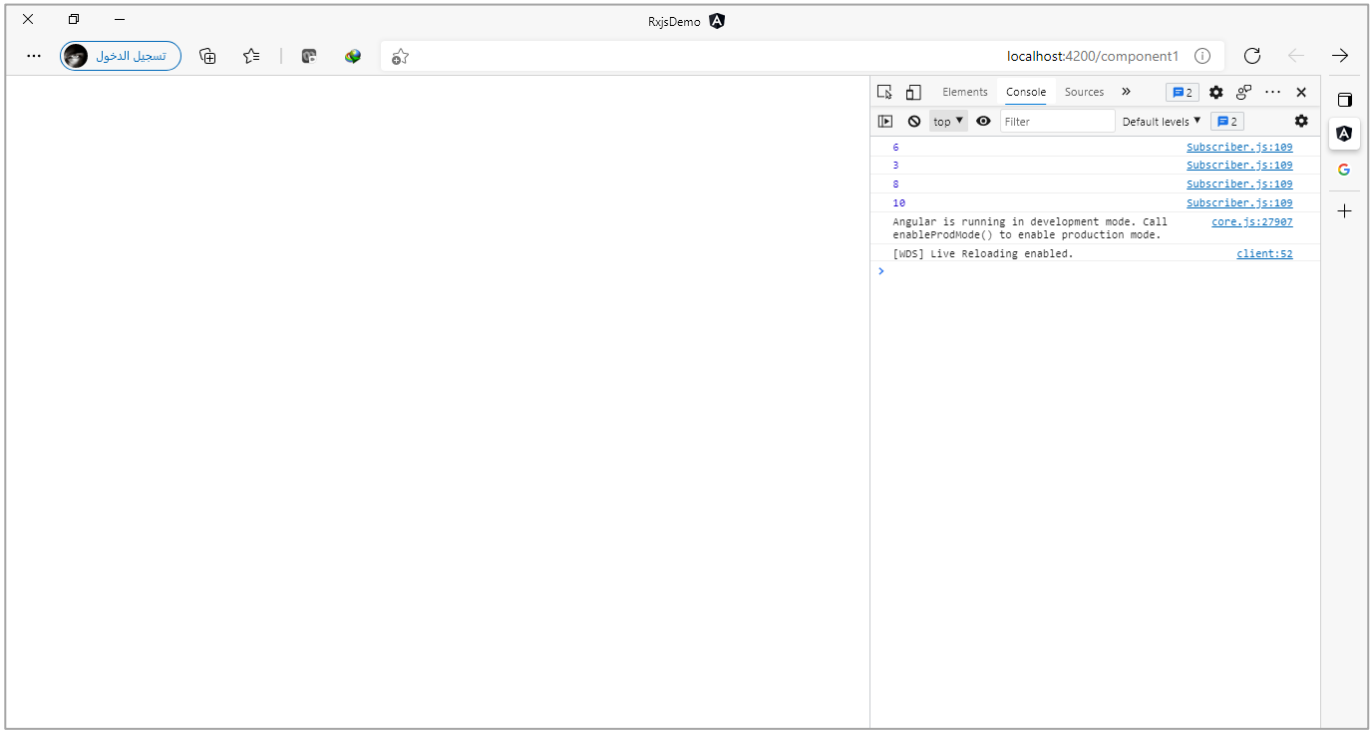
وهذه الدوال عكس دوال taking حيث هناك كُنا نأخذ القيم في حال تحقق الشرط ونتجاهل باقي القيم عند اول قيمة لم تحقق الشرط، اما هنا فتقوم بتجاهل القيم Skip Values في حال **تحقق الشرط** ونقرأ القيم عند اول قيمة لم تحقق الشرط وما يليها من قيم.

skip() -1-4-2-3:

هذه الدالة تستقبل قيمة رقمية ويتم تجاهل قيم الObservable بحسب العدد الذي تم تمريره لهذه الدالة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { skip } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 6, 3, 8, 10)
      .pipe(
        skip(2),
      )
      .subscribe(console.log);
  }
}
```



3-2-4-2-skipUntil() :

اما هذه الدالة فهي عكس الدالة takeUntil لذلك سوف نقوم بتطبيق المثال الذي قمنا بتطبيقه على الدالة السابقة مع تعديل في ملفي الTemplate والClass في الComponent ذو الاسم sub، كالتالي:

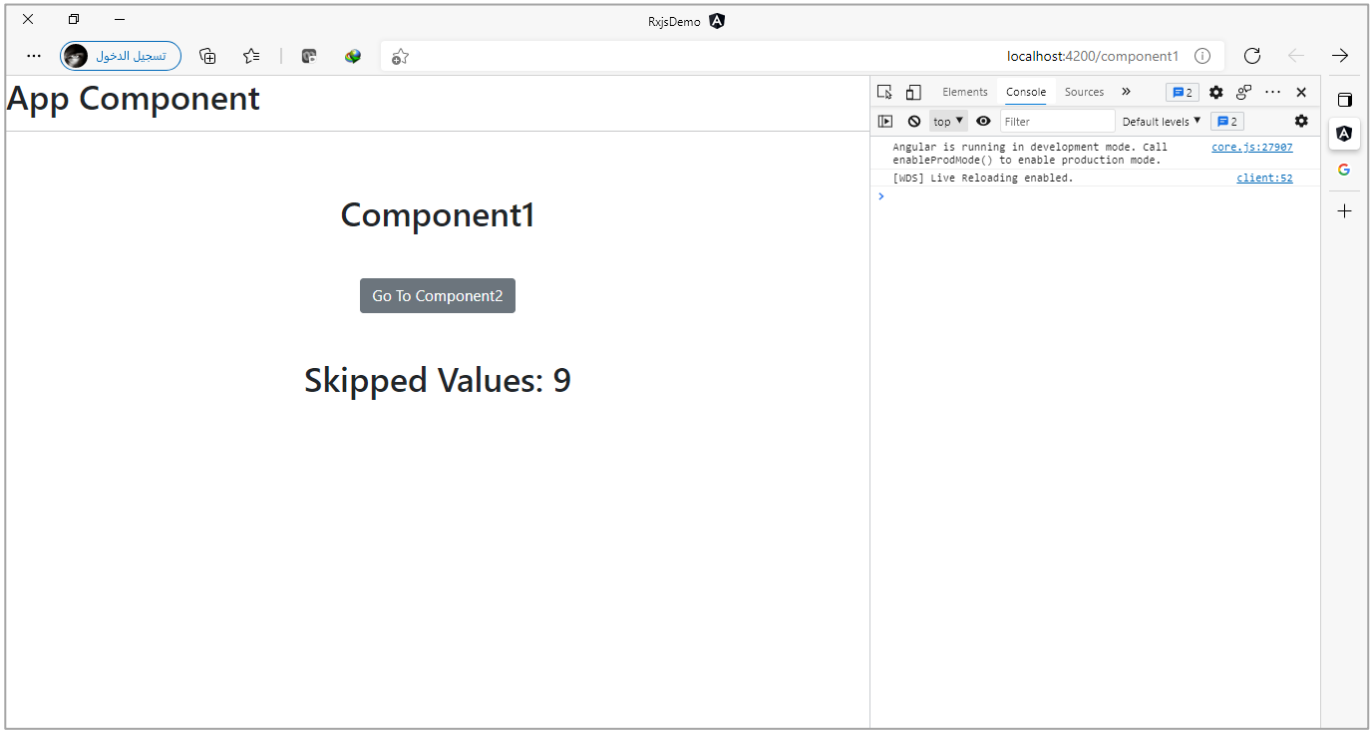
ملف sub.component.html

```
<div style="display: flex; flex-direction: column; align-items: center">
  <h1 class="m-5">Component1</h1>
  <a class="btn btn-secondary" routerLink="/component2">Go To Component2</a>
  <h1 class="m-5">Skipped Values: {{ values }}</h1>
</div>
```

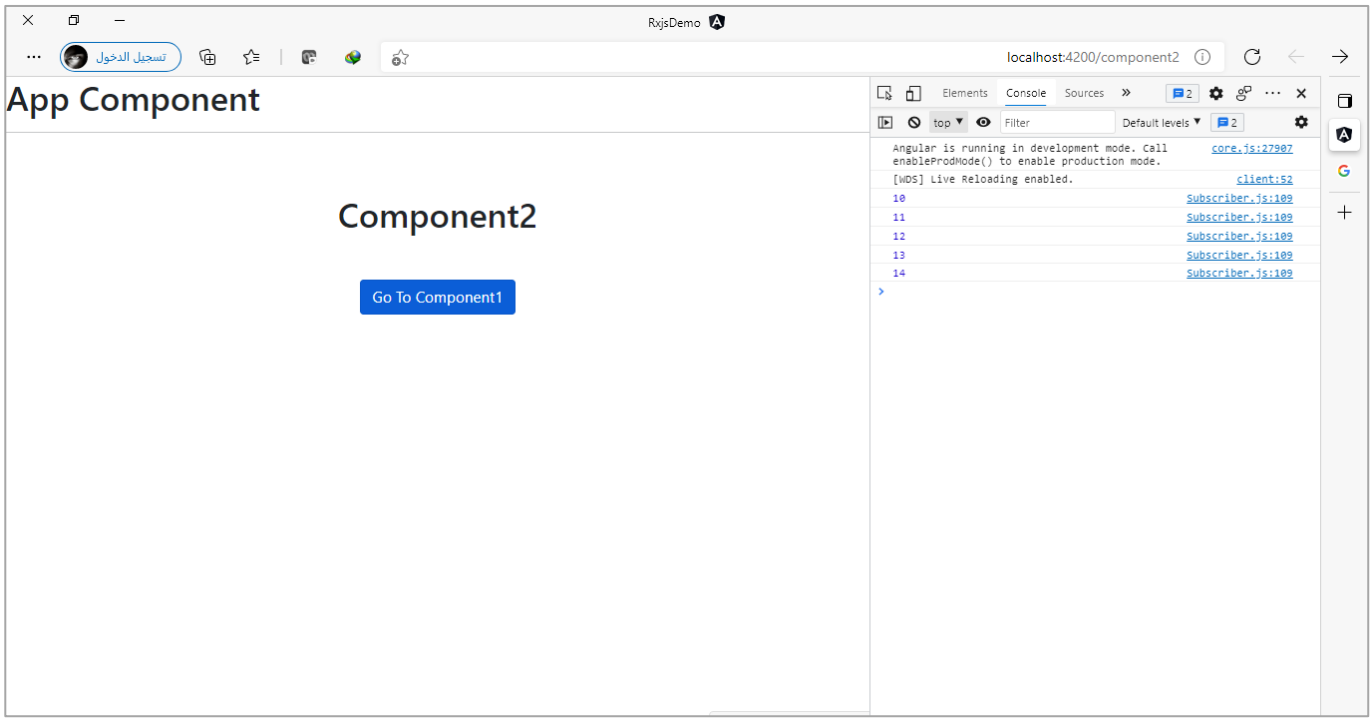
ملف sub.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
import { skipUntil, tap } from 'rxjs/operators';
import { TestService } from '../test.service';

@Component({
  selector: 'app-sub',
  templateUrl: './sub.component.html',
  styleUrls: ['./sub.component.scss'],
})
export class SubComponent implements OnInit, OnDestroy {
  private notifier = new Subject();
  values = 0;
  constructor(private test: TestService) {}
  ngOnInit(): void {
    this.test.timer
      .pipe(
        tap((val: number) => (this.values = val)),
        skipUntil(this.notifier)
      )
      .subscribe(console.log);
  }

  ngOnDestroy(): void {
    this.notifier.next('');
  }
}
```

اما باقي ملفات مثال الدالة takeUntil فتبقى كما هي بدون تعديل، ولنشاهد النتيجة كالتالي:



في البداية ظهرت لنا القيم التي تم عمل لها Skipped وقد اظهرتها هنا لتوضيح فقط حيث استخدمت الدالة tap (سنشرحها لاحقاً بإذن الله) لإظهار هذه القيم، الآن لنضغط على الزر ونرى النتيجة:



نلاحظ عندما تم عمل emit لقيمة ما تم قراءة القيم وتجاهل القيم السابقة.

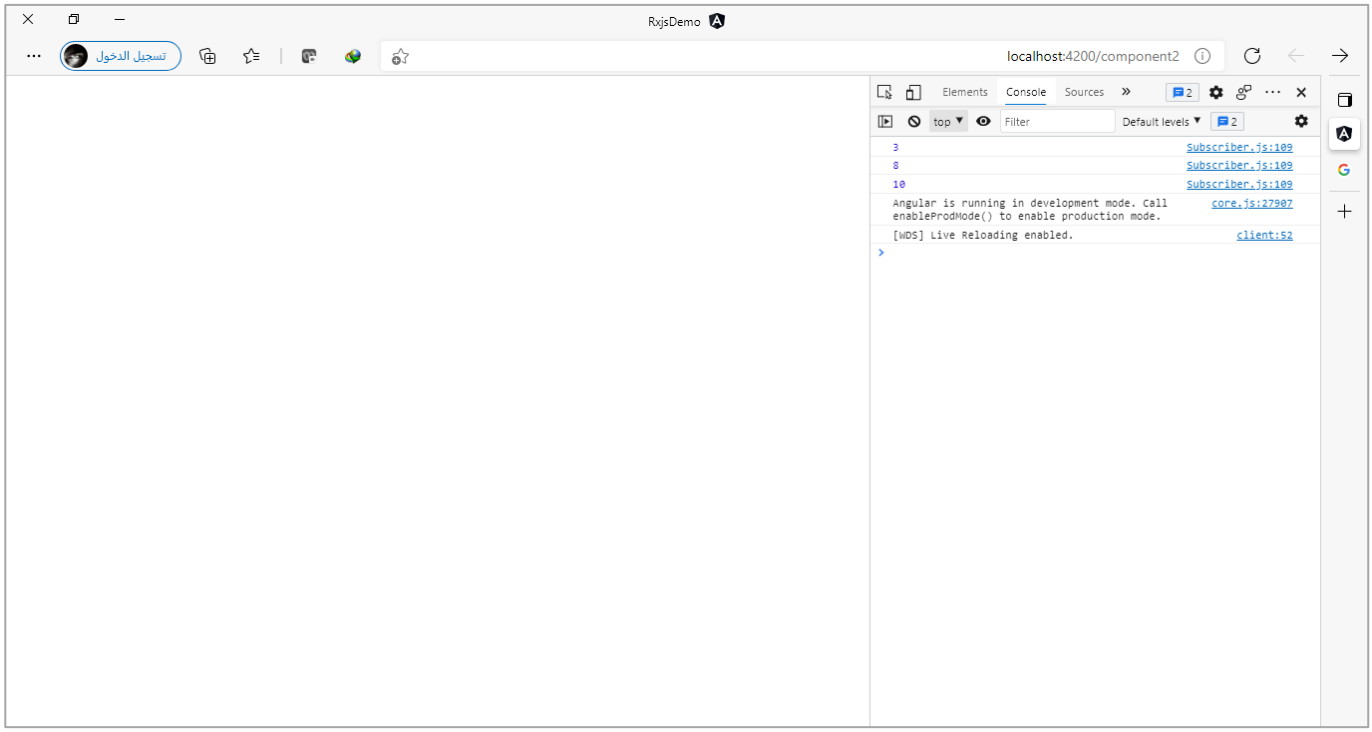
skipWhile() -3-4-2-3:

اما هذه الدالة فتقوم بعمل skip للقيم طالما الشرط متحقق وعندما لم يتحقق الشرط تقرأ القيمة التالية مع القيمة التي لم تحقق الشرط، وهي مشابهه لدالة takeWhile حيث تستقبل دالة Function، كالتالي:

app.component.ts ملف

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { skipWhile } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 6, 3, 8, 10)
      .pipe(
        skipWhile((val: number) => {
          return val % 2 === 0;
        })
      )
      .subscribe(console.log);
  }
}
```

والنتيجة:



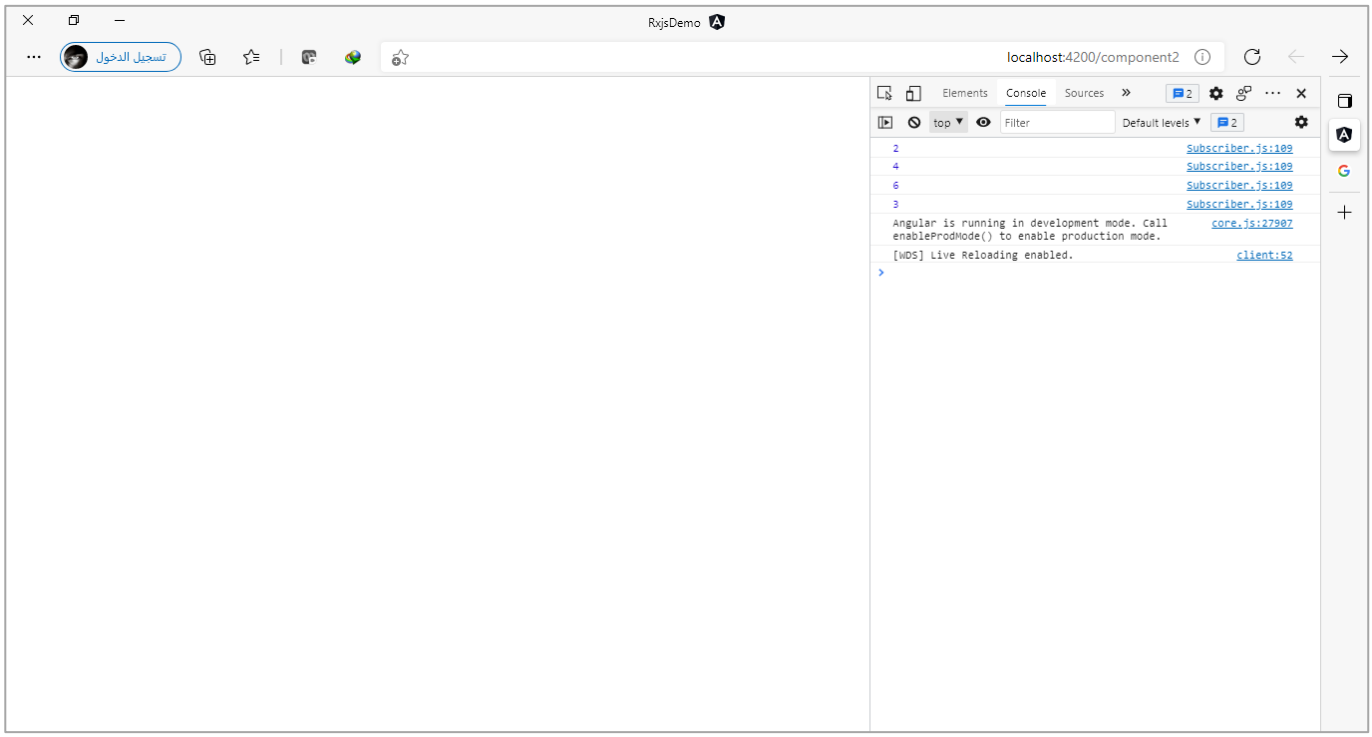
skipLast() -4-4-2-3:

هذه الدالة هي مشابهه لدالة skip وتختلف عنها انها تقوم بعمل تجاهل لآخر قيم الObservable بحسب العدد الذي تم تمريره لها، كالتالي:

app.component.ts ملف

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { takeLast } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 6, 3, 8, 10)
      .pipe(
        skipLast(2)
      )
      .subscribe(console.log);
  }
}
```

والنتيجة:



كما نلاحظ تم تجاهل آخر قيمتين.

3-2-5- Distinctness Operators:

والمقصود فيها الدوال التالية:

- distinct()

- distinctUntilChanged()
- distinctUntilKeyChanged()

هذه الدوال تتشارك فيما بينها بوظيفة فلترة وترشيح قيم الObservable من خلال حذف القيم المتشابهة، وتختلف في آليه هذه الفلترة وSyntax الخاص بكل دالة.

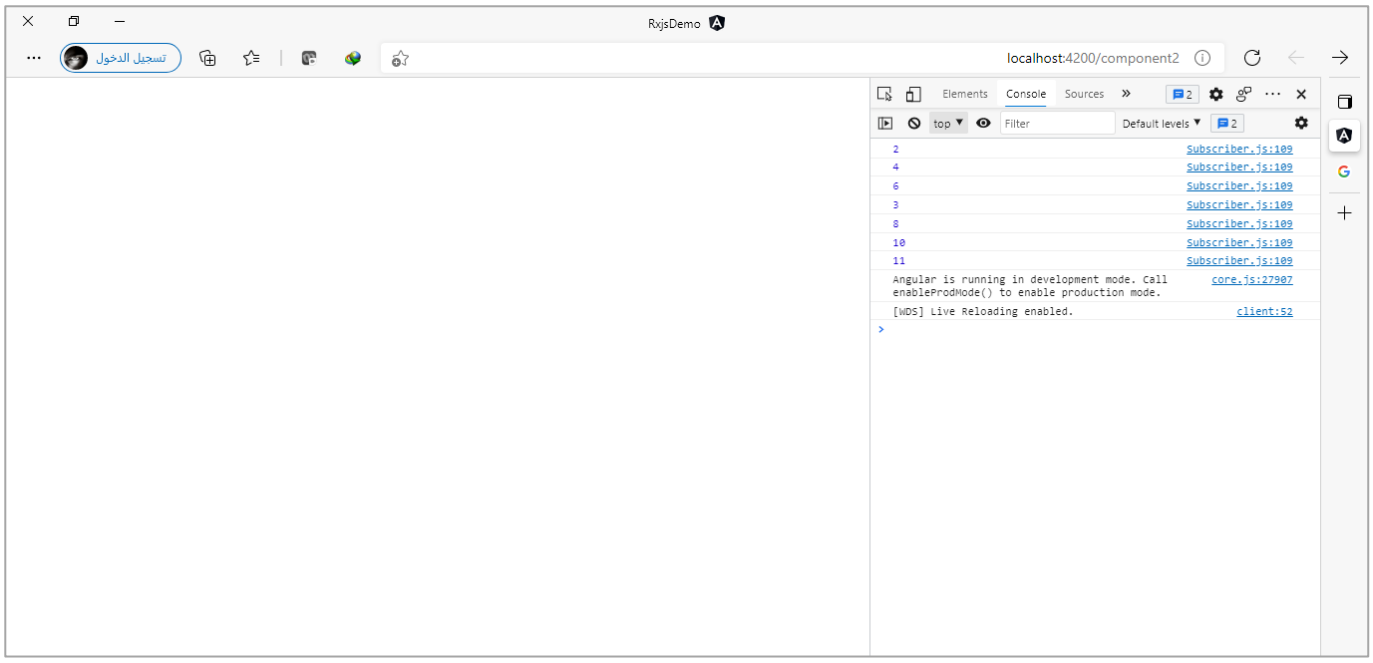
3-2-5-1- distinct():

هذه الدالة تقوم بقراءة جميع القيم في الObservable واي قيم متشابهة سيتم حذفها مباشرة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { distinct } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 4, 4, 6, 3, 3, 8, 10, 11, 11, 11, 11)
      .pipe(distinct())
      .subscribe(console.log);
  }
}
```

كما نلاحظ لدينا Observable يحتوي على قيم متشابهة لذلك استخدمنا الدالة distinct لترشيح وحذف القيم المتكررة.



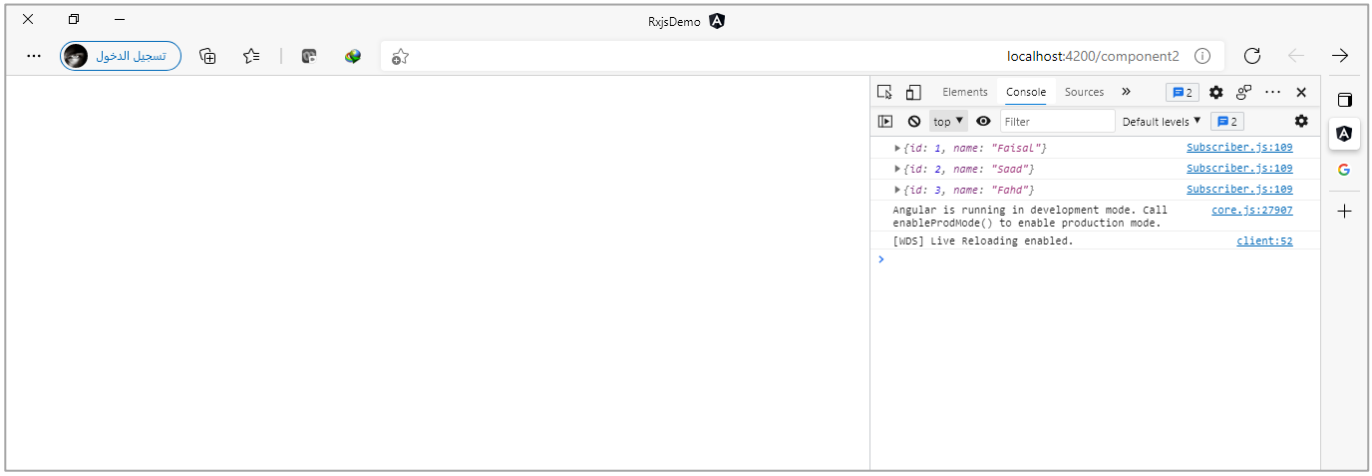
هذا في حال كانت القيم من نوع واحد كما في المثال السابق ولكن ماذا لو كانت لدينا قيم مختلفة من هذا الObservable ونُريد ان نحذف القيم من نوع من أنواع هذه البيانات، كأن يكون لدينا مصفوفة من الكائنات وكُل كائن يحتوي على مجموعة من الخصائص، ونُريد مثلاً ان نحذف القيم المتكررة في خاصية معينة من كل كائن.

وللقيام بذلك نستطيع استخدام دالة distinct ومن ثم نمرر لها باراميتر اختياري عبارة عن دالة function وهذه الدالة تمثل كل كائن ومن ثم نُعيد الخاصية التي نُريد ان نعمل لها فلترة ونحذف قيمها المتكررة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';
import { distinct } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    from([
      { id: 1, name: 'Faisal' },
      { id: 2, name: 'Saad' },
      { id: 3, name: 'Fahd' },
      { id: 1, name: 'Khalid' },
    ])
      .pipe(distinct((values: any) => values.id))
      .subscribe(console.log);
  }
}
```

كما نلاحظ مررنا دالة Function إلى هذه الدالة ومن ثم استخدمنا الطريقة المختصرة لعمل إعادة return للخاصية id التي نُريد ان حذف القيم المتكررة بها.



3-2-5-2- distinctUntilChanged():

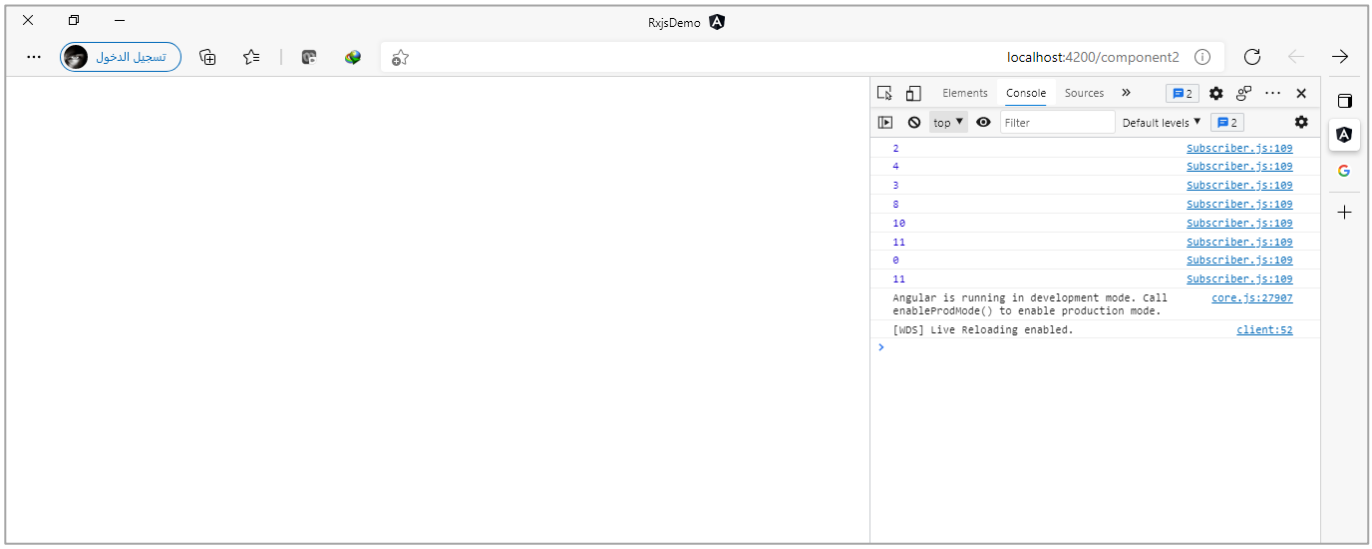
هذه الدالة تقوم بحذف القيم المتشابهة لكن بشرط ان تكون متتالية، بحيث لو كان لدينا الاعداد [1, 2, 2, 3, 2, 2] فان النتيجة ستكون [1, 2, 3, 2] بعكس الدالة distinct التي تقوم بحذف جميع القيم المتشابهة بحيث تكون النتيجة لو استخدمنا الدالة distinct [1, 2, 3].

وفي الغالب يتم استخدام هذه الدالة في اجراء عمليات البحث من خلال input حيث عندما يقوم المستخدم بكتابة حروف معينة فنستطيع استخدام هذه الدالة لتعامل مع هذه الحالات، وسوف نتطرق لها لاحقاً بإذن الله عندما نتطرق إلى دالة debounceTime.

وألان لنستعرض مثال على دالة distinctUntilChanged، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(2, 4, 4, 4, 6, 3, 3, 8, 10, 11, 11, 0, 11, 11)
      .pipe(distinctUntilChanged())
      .subscribe(console.log);
  }
}
```

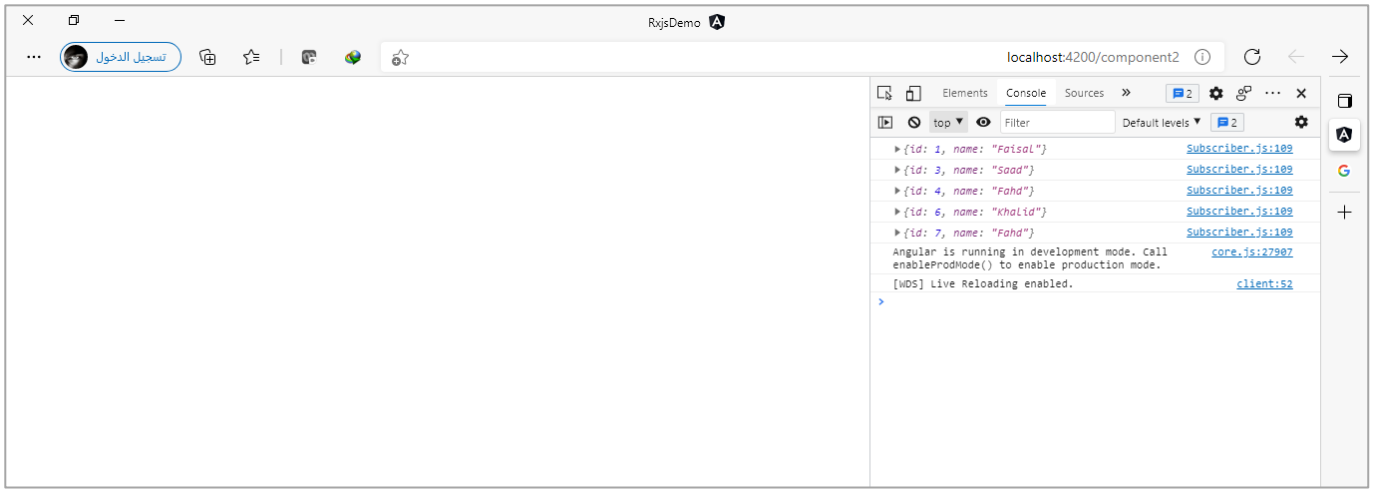


ونفس ما قيل في الدالة distinct يُقال هنا من حيث إذا كان لدينا خاصية من ضمن كائن نستطيع حذف القيم المتكررة ولكن هنا syntax مختلف قليلاً حيث نمرر دالة Function على شكل بارامتر لـdistinctUntilChanged وهذه Function نمرر لها بارامترين الأول يمثل القيمة الأولى والثاني يمثل القيمة الثانية التي تليها، بحيث نعمل مقارنة بينها، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    from([
      { id: 1, name: 'Faisal' },
      { id: 2, name: 'Faisal' },
      { id: 3, name: 'Saad' },
      { id: 4, name: 'Fahd' },
      { id: 5, name: 'Fahd' },
      { id: 6, name: 'Khalid' },
      { id: 7, name: 'Fahd' },
    ])
      .pipe(
        distinctUntilChanged((value1: any, value2: any) => {
          return value1.name === value2.name;
        })
      )
      .subscribe(console.log);
  }
}
```

كما تلاحظ قمنا بالمقارنة بين خاصية name لكل كائن بحيث نلغي الأسماء المتشابهة **المتتالية**، وتكون النتيجة:



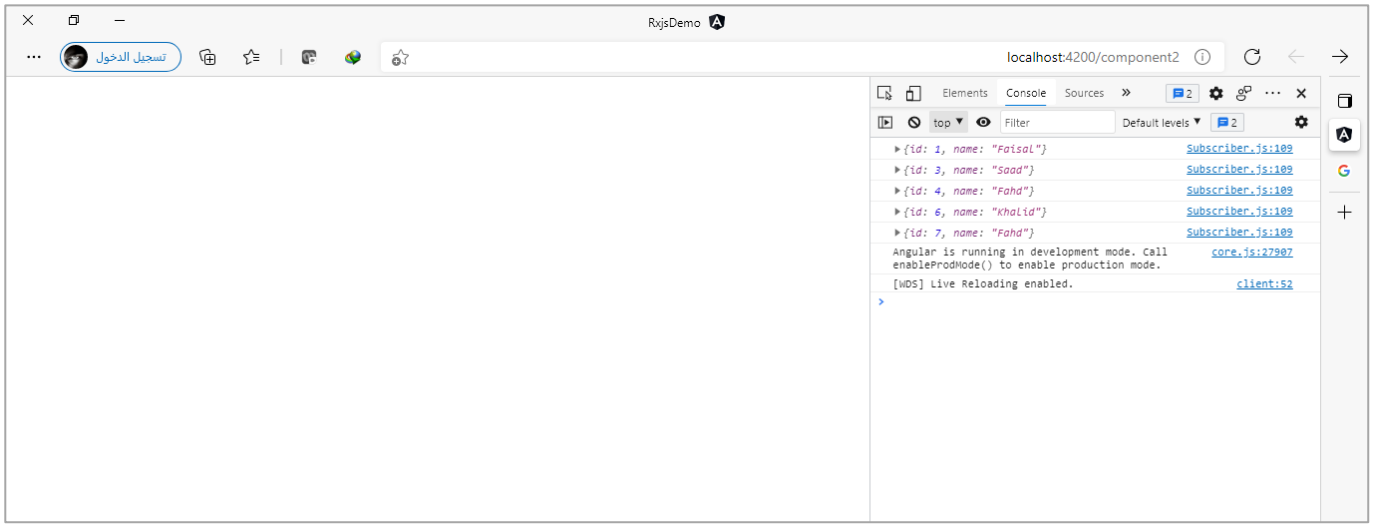
3-5-2-3- distinctUntilKeyChanged():

اما هذه الدالة فهي مشابهة لدالة السابقة ولكن بي Syntax أفضل وأسهل، فهنا نمرر لدالة نص string يمثل الخاصية التي نُريد ان نعمل مقارنة بينها وحذف المتكرر منها بشرط ان تكون القيم متتالية، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { from } from 'rxjs';
import { distinctUntilKeyChanged } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    from([
      { id: 1, name: 'Faisal' },
      { id: 2, name: 'Faisal' },
      { id: 3, name: 'Saad' },
      { id: 4, name: 'Fahd' },
      { id: 5, name: 'Fahd' },
      { id: 6, name: 'Khalid' },
      { id: 7, name: 'Fahd' },
    ])
      .pipe(distinctUntilKeyChanged('name'))
      .subscribe(console.log);
  }
}
```

كما نلاحظ مررنا فقط الخاصية name لهذه الدالة فقط ولم نمرر Function، كما هو الحال في الدالة السابقة.



3-2-6-3- Timing Operators*:

والمقصود فيها الدوال التالية:

- debounceTime()
- auditTime()
- throttleTime()

جميع هذه الدوال تشترك فيما بينها بوظيفة تأخير استقبال قيم الObservable لفترة زمنية معينة يتم تمريرها لهذه الدوال على شكل بارامتر بالميلي ثانية، لكي نقوم بفلترة هذه القيم، وتختلف بآلية هذا التأخير وطريقة استقبال القيم.

3-2-6-3- 1-debounceTime():

لعل هذه الدالة من أشهر هذه الدوال وتُستخدم في العادة في عمليات البحث المباشر في قاعدة البيانات او في الاقتراحات autocomplete التي تتصل بقاعدة البيانات لجلب وعرض هذه الاقتراحات، كما هو الحال في محرك البحث Google حيث عندما نقوم بكتابة حرف معين سيتم الاتصال بسيرفر معين ويظهر لك اقتراحات، او مثلاً لو كان لديك بريد الكتروني وتُريد التأكد مباشرة من خلال مربع النص والمستخدم يكتب في مربع النص بحيث نبحث في قاعدة البيانات ونظهر رسالة للمستخدم بأن هذا البريد مُتاح من عدمه.

وهذه الحالات وغيرها لا نُريد مع كُل ضغطة على لوحة المفاتيح ان يتم الاتصال بقاعدة البيانات وانما نُريد ان نأخذ القيم من مربع النص التي كتبها المستخدم ومن ثم نقوم بتأخير أرسالها لفترة زمنية معينة كأن تكون مثلاً 500 ميلي ثانية (نص ثانية) ومن ثم يتم ارسال هذه القيم لسيرفر، لذاك أتت لدينا هذه الدالة التي تُتيح لنا القيام بهذا الأمر بكل بساطة.

اما طريقتها بتأخير واستقبال البيانات فتتم من خلال تمرير بارامتر يُمثل الفترة الزمنية التي نُريد ان يتأخر فيها ارسال القيم كأن تكون مثلاً 1000 ميلي ثانية، ومن ثم تقوم باستقبال اول قيمة في مربع النص ولتكن مثلاً a ومن ثم تنتظر ثانية واحدة فإذا تم ارسال قيمة أخرى خلال هذه الثانية فإنها ستقوم بقراءة هذه القيمة ودمجها مع القيم السابقة ان وجدت وايضاً تقوم بإعادة الوقت وتنتظر ثانية أخرى، وفي هذه الثانية ايضاً تنتظر فإذا تم ارسال قيمة خلال تثوم بإعادة الخطوة السابقة وإذا لم يتم ارسال القيمة ستقوم بارسال هذه القيم، وتنتظر القيم الجديدة وهكذا إلى ان تنتهي من جميع القيم.

ولتوضيح لنُعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { interval } from 'rxjs';
import { debounceTime, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  form: FormGroup = new FormGroup({
    name: new FormControl(),
  });

  counter = 0;

  constructor() {}

  ngOnInit(): void {
    interval(1000)
      .pipe(
        tap(() => {
          this.counter++;
          if (this.counter === 5) {
            this.counter = 0;
          }
        })
      )
      .subscribe();

    this.form.valueChanges
      .pipe(
        tap(() => (this.counter = 0)),
        debounceTime(4000)
      )
      .subscribe((data) => console.log(data));
  }
}
```


كنوع من التغيير قمنا في النقطة 1 بالوصول إلى مربع النص باستخدام Angular Forms بدلاً من @ViewChild ولمعرفة اعمق عن Angular Forms الرجاء مراجعة كتابي الذي يحمل نفس الاسم وهو جزء من هذه السلسلة.

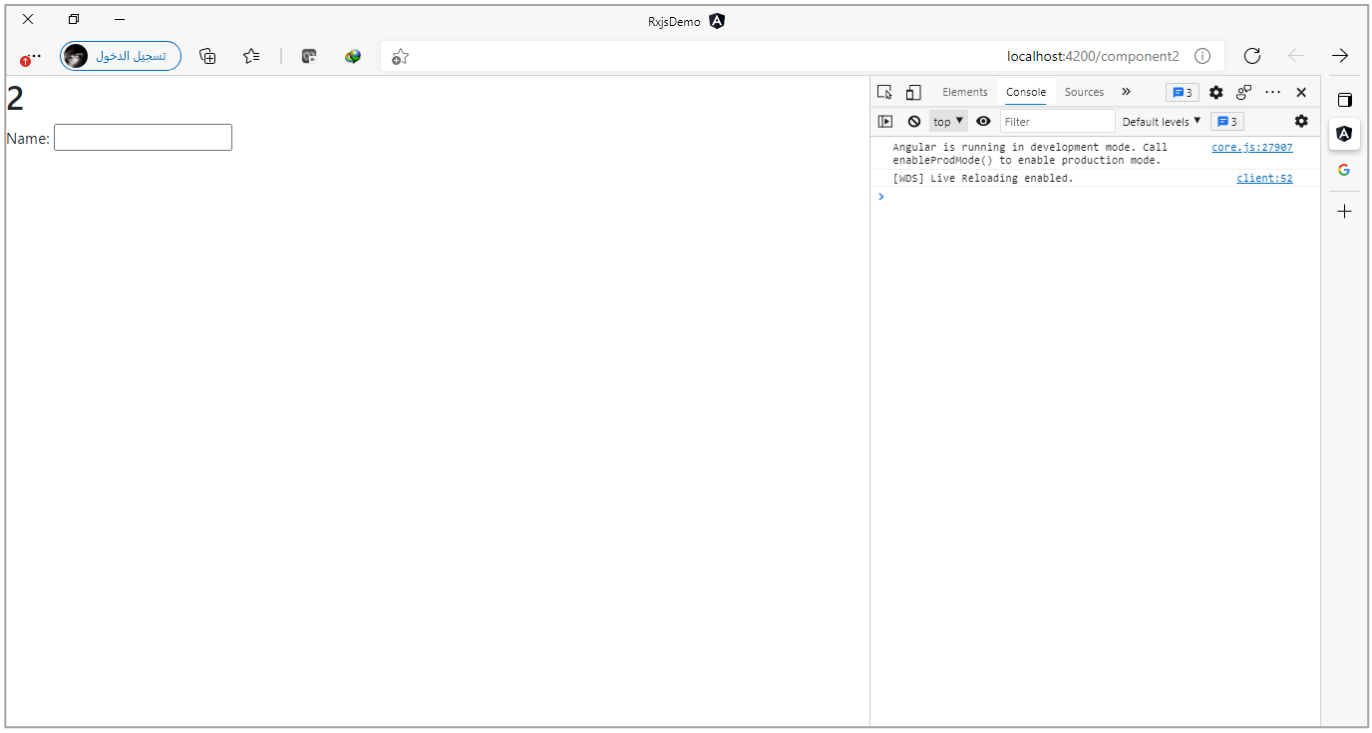
اما النقطة 2 و3 فهذه الاسطر للتوضيح وليس لها علاقة من قريب او بعيد بالدالة debounceTime، والغرض منها هو عداد يقوم بالعد من 1 إلى 4 ثواني وعند الوصول إلى اربع ثواني يرجع العد مره أخرى ويعد اربع ثواني وفي حال كتابة أي شيء في مربع النص يتم ايضاً تصفير العداد والبدء مجدداً والسبب لأن سلوك الدالة debounceTime تقوم بتصفير العداد والبدء من جديد عندما تأتي لها أي قيمة لذلك حاولت محاكاة هذا الامر من خلال هذه الاسطر البرمجية.

اما النقطة الأخيرة، فهذه الاسطر هي التي تختص بهذه الدالة ما عدا الtap حيث قمنا بتمرير 4000 ميلي ثانية وهذا معناه ان هذه الدالة سوف تستقبل القيمة من الObservable ومن ثم تنتظر 4 ثواني فإذا تم ارسال قيمه لها خلال هذه الأربع ثواني ستقوم بتصفير العداد والبدء من جديد والانتظار اربع ثواني أخرى مع دمج القيم الجديدة مع القديمة وهكذا إلى ان تنتهي.

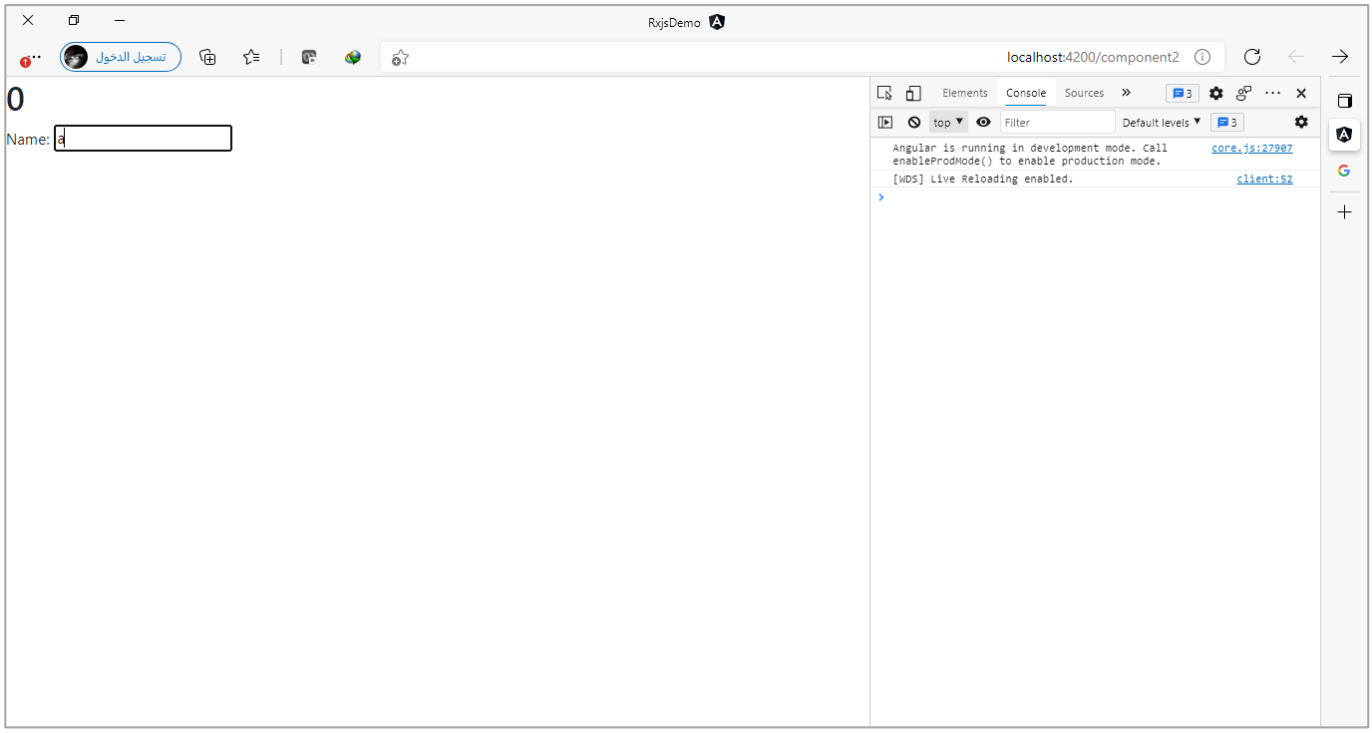
app.component.html ملف

```
<div>
  <h1>{{ counter }}</h1>
</div>
<form [formGroup]="form">Name: <input formControlName="name" /></form>
```

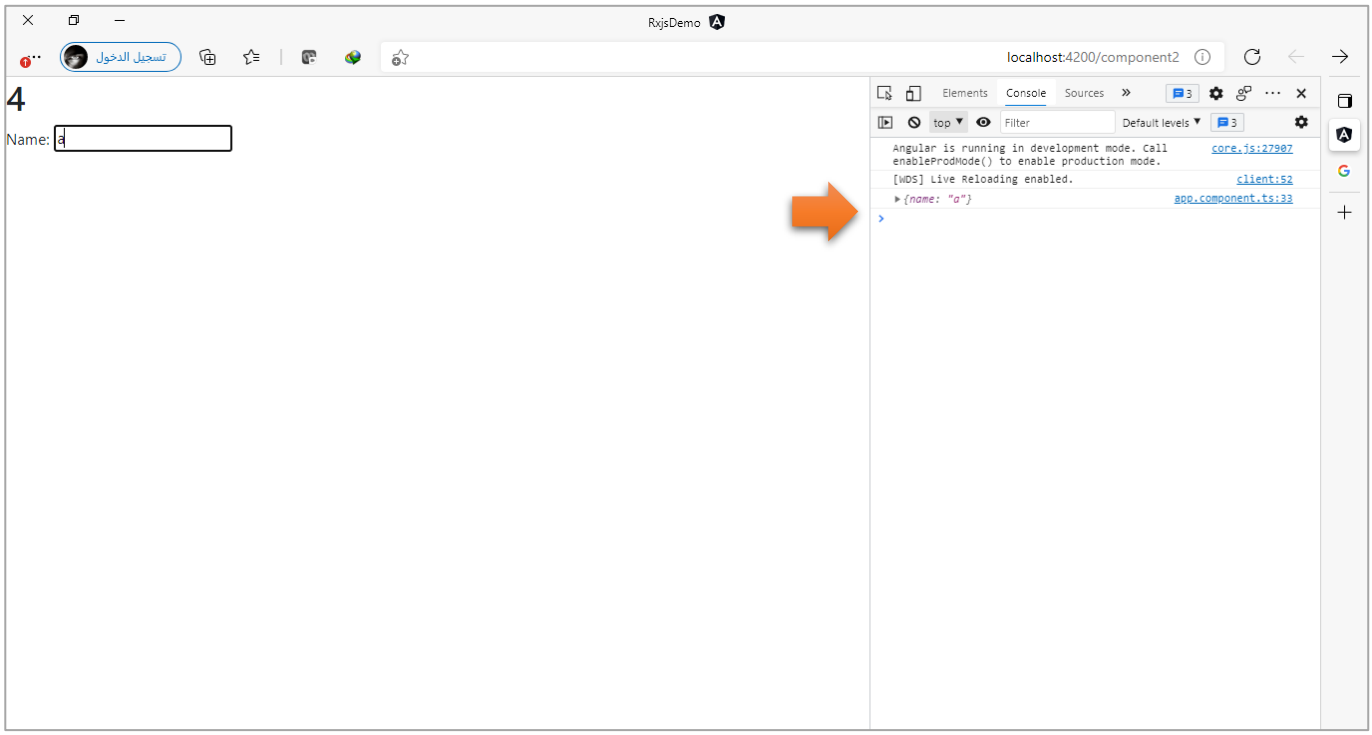
اما هنا فقد قمنا فقط بربط أداة مربع النص مع الاسم البرمجي الخاص بها، وايضاً عملنا Binding للعداد لكي نستطيع مشاهدته في المتصفح، كالتالي:



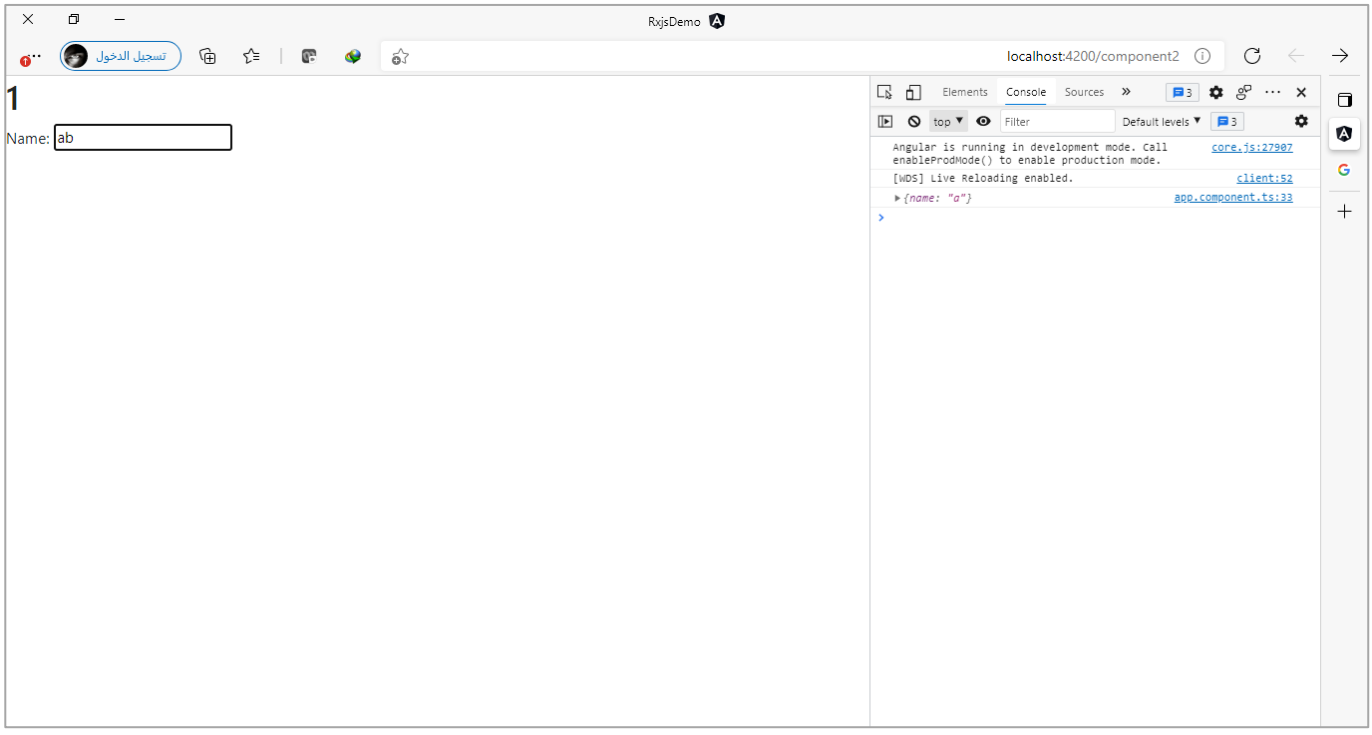
كما نلاحظ العداد مستمر في العد ومربع النص فارغ، الآن لنقوم بكتابة حرف واحد في مربع النص ولنرى النتيجة:



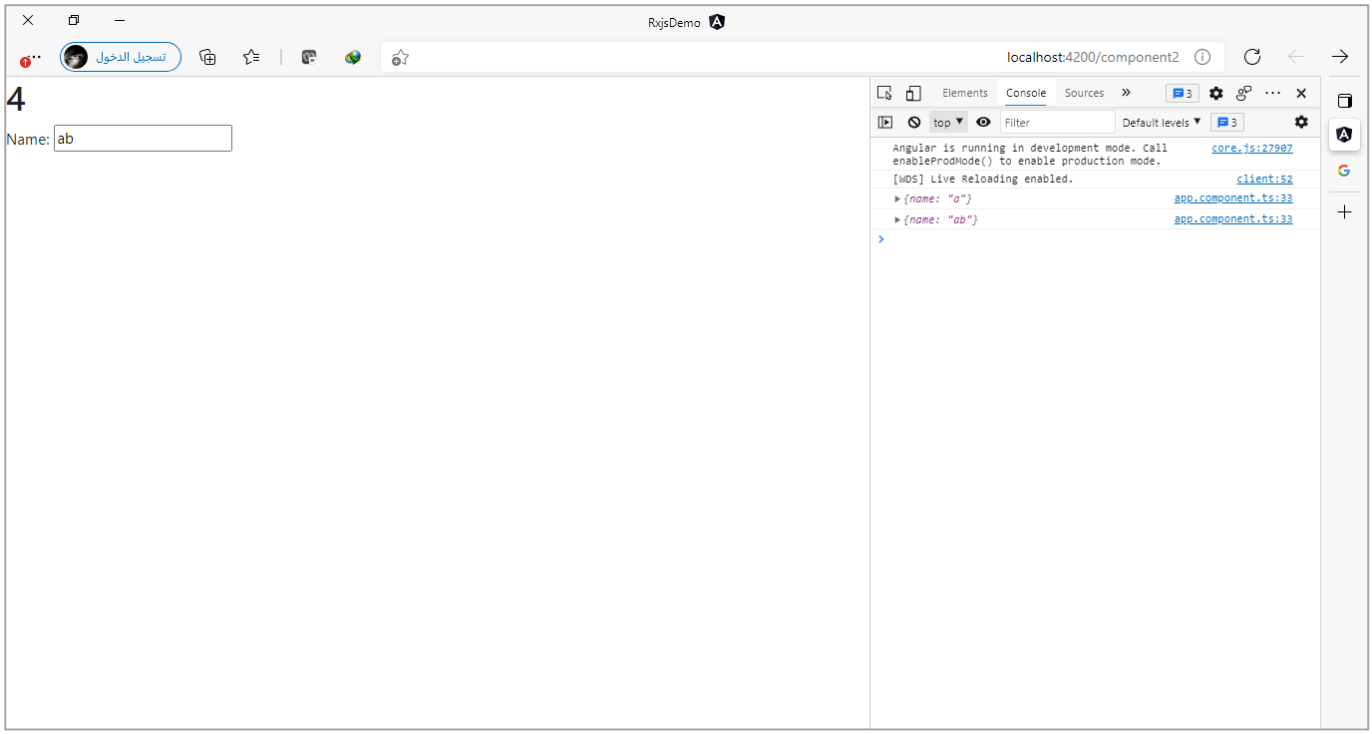
نلاحظ تم تصفير العداد لأننا نُريد عمل محاكاة لعمل الدالة debounceTime كما أشرت سابقاً، وذلك لأننا عملنا subscribe لأداة مربع النص لكي نقرأ أي تغيير في قيمه على شكل Observable، ومررنا هذه القيم على الدالة debounceTime، والآن بعد مرور اربع ثواني سيتم ارسال هذه القيم وعرضها في console، كالتالي:



والان لنقوم بكتابة حرف آخر، وهذا معناه ان قيمة جديدة سوف تُرسل من خلال هذا الObservable وكما قلنا سابقاً عند وجود قيمة جديدة ستقوم الدالة debounceTime بإعادة تصفير الوقت بدأ العد من جديد، وعند انتهاء الوقت ولم يتم ارسال أي قيمة ستقوم بإرسال القيم إلى console، ويتضح هذا جلياً في المحاكاة التي قمنا بعملها عن طريق العداد حيث عند كتابة أي قيمة سيتم تصفير هذا العداد، كالتالي:



وبعد مرور 4 ثواني سيتم عرض القيم في console، كالتالي:



اما الآن لنعطي مثال آخر أكثر واقعية، ومصدر المثال على الرابط التالي: Angular 9|8 Add Debounce Time using RxJS 6 to Optimize Search Input for API results from server « Freaky Jolly وقد أجريت عليه بعض التعديلات لتسهيل ولكي لا يتشتت انتباهك إلى نقاط أخرى ويتم التركيز فقط على ما نُريد شرحه.

وتقوم فكرة المثال على وجود مربع نص للبحث عن الأفلام، وعند كتابة أي قيمة في مربع النص سيتم الاتصال في قاعدة بيانات تابعة لموقع IMDB بواسطة API معين للبحث وفق القيم المدخلة في مربع النص.

مع العلم ان البحث سيكون مباشر وهذا يعني أي قيمة او أي تعديل يتم في مربع النص سيتم الاتصال بقاعدة البيانات للبحث عنه بشكل مباشر.

والآن لنستعرض الكود، كالتالي:

ملف app.component.html

```
<div style="margin: 50px">
  <div class="container" style="text-align: center">
    <div class="row">
      <div class="col-12 text-center">
        <h1>Angular Search using Debounce in RXJS 6</h1>

        <input
          type="text"
          #movieSearchInput
          class="form-control"
          placeholder="Type any movie name"
        />
      </div>
```

```
    </div>
    <div class="row" *ngIf="isSearching">
      <div class="col-12 text-center">
        <h4>Searching ...</h4>
      </div>
    </div>
    <div class="row">
      <ng-container
        *ngIf="apiResponse['Response'] == 'False'; else elseTemplate"
      >
        <div class="col-12 text-center">
          <div class="alert alert-danger" role="alert">
            {{ apiResponse["Error"] }}
          </div>
        </div>
      </ng-container>

      <ng-template #elseTemplate>
        <div class="col-3" *ngFor="let movie of apiResponse['Search']">
          <div class="card" style="margin: 5px">
            <img class="card-img-top" src="{{ movie['Poster'] }}" />
            <div class="card-body">
              <h5 class="card-title">{{ movie["Title"] }}</h5>
              <p class="card-text">Year: {{ movie["Year"] }}</p>
            </div>
          </div>
        </div>
      </ng-template>
    </div>
  </div>
</div>
```

اولاً ملف template لم اقم بأي تعديل عليه، وتركته كما هو موجود في المثال الأصلي، اما ملف class فقد أجريت عليه التعديلات التالية: مع العلم انه في ملف template تم استخدام مكتبة Bootstrap لتنفيذ التنسيقات.

app.component.ts ملف

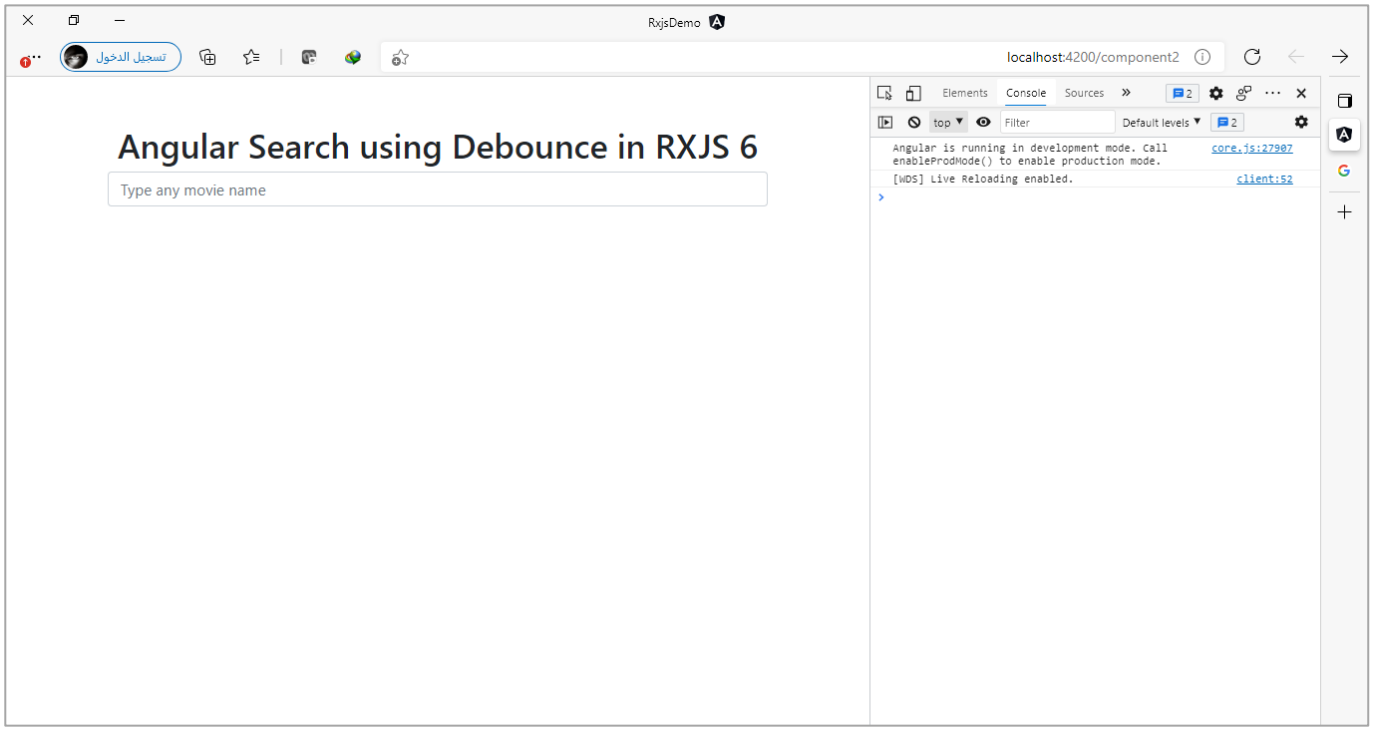
```
import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';
import { debounceTime, map, distinctUntilChanged, switchMap, tap, filter } from 'rxjs/operators';
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('movieSearchInput', { static: true }) movieSearchInput: ElementRef;
```

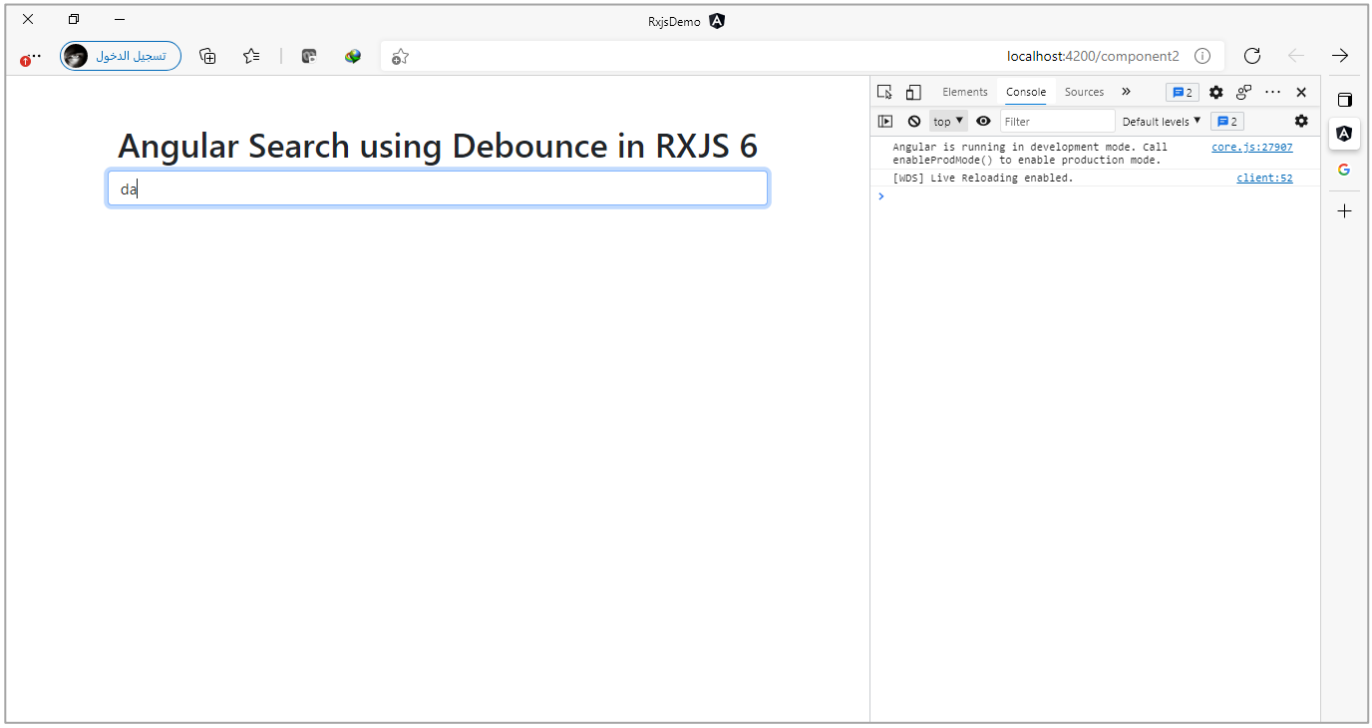
```
  apiResponse: any[] = [];
  isSearching = false;

  constructor() {}

  ngOnInit(): void {
    fromEvent(this.movieSearchInput.nativeElement, 'keyup')
      .pipe(
        // get value
        map((event: any) => event.target.value),

        // if character length greater then 2
        filter((res: any) => res.length > 2),

        // show searching message
        tap(() => (this.isSearching = true)),

        // Time in milliseconds between key events
        debounceTime(1000),

        // If previous query is different from current
        distinctUntilChanged(),

        // connect db to get data by API
        switchMap((textSearch: any) => {
          return ajax({
            url: `http://www.omdbapi.com/?s=${textSearch}&apikey=e8067b53`,
            method: 'get',
            queryParams: textSearch,
            crossDomain: true,
          });
        }),
        // return response property(The Data)
        map((res: any) => res.response)
      )
      // subscription for response
      .subscribe({
        next: (response: any) => {
          this.apiResponse = response;
          this.isSearching = false;
        },
        error: (error) => {
          this.isSearching = false;
        },
      });
  }
}
```

بطبيعة الحال لن استعرض جميع الاسطر البرمجية لعدة أسباب منها ان هذه الأوامر يُفترض بالمتعلم ان يكون مُلم بها او انها خارج نطاق هذا الكتاب مثل ajax والتي تُستخدم للاتصال بقواعد البيانات عن طريق API لأننا في عالم Angular

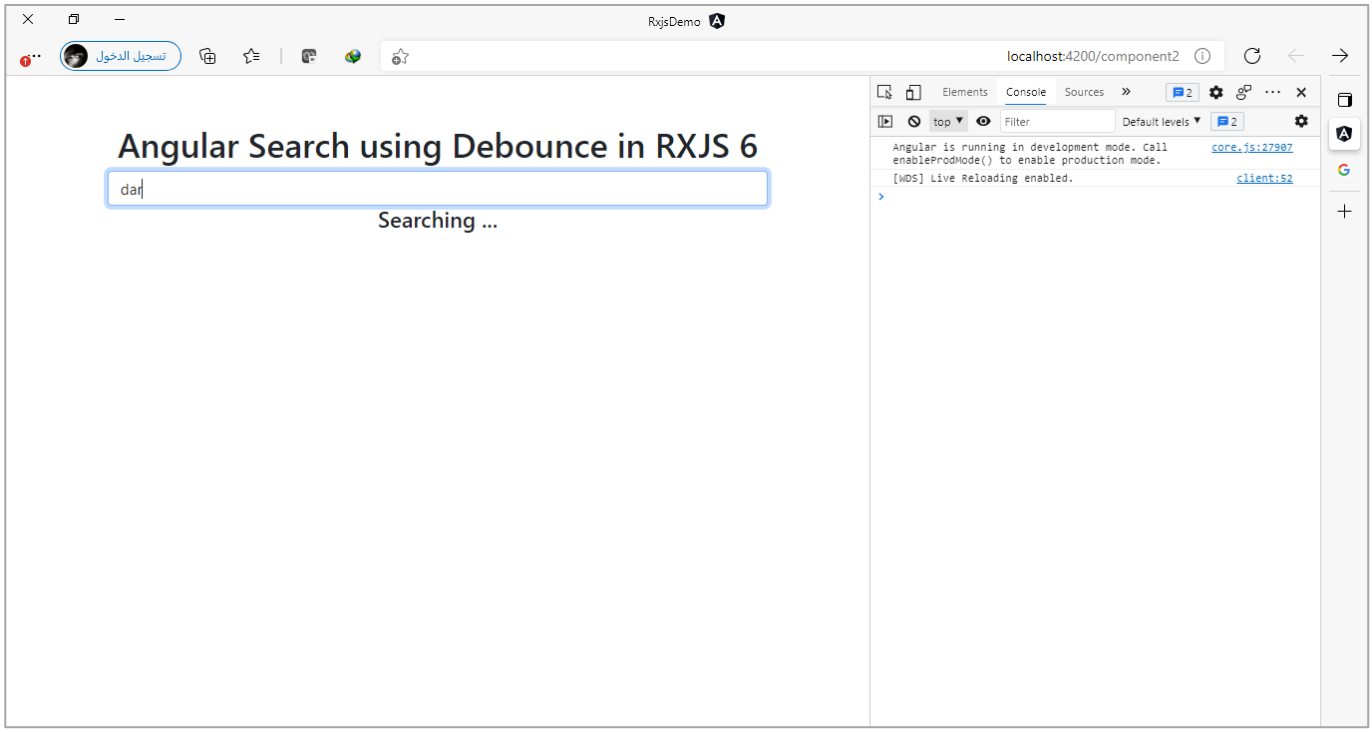
لا نستخدم هذه الطريقة وانما لدينا Module جاهز اسمه HttpClient يُسهل لنا التعامل مع هذه التقنيات، وسوف أُفرد له فصل كامل بإذن الله بهذا الكتاب نشرحه بالتفصيل.

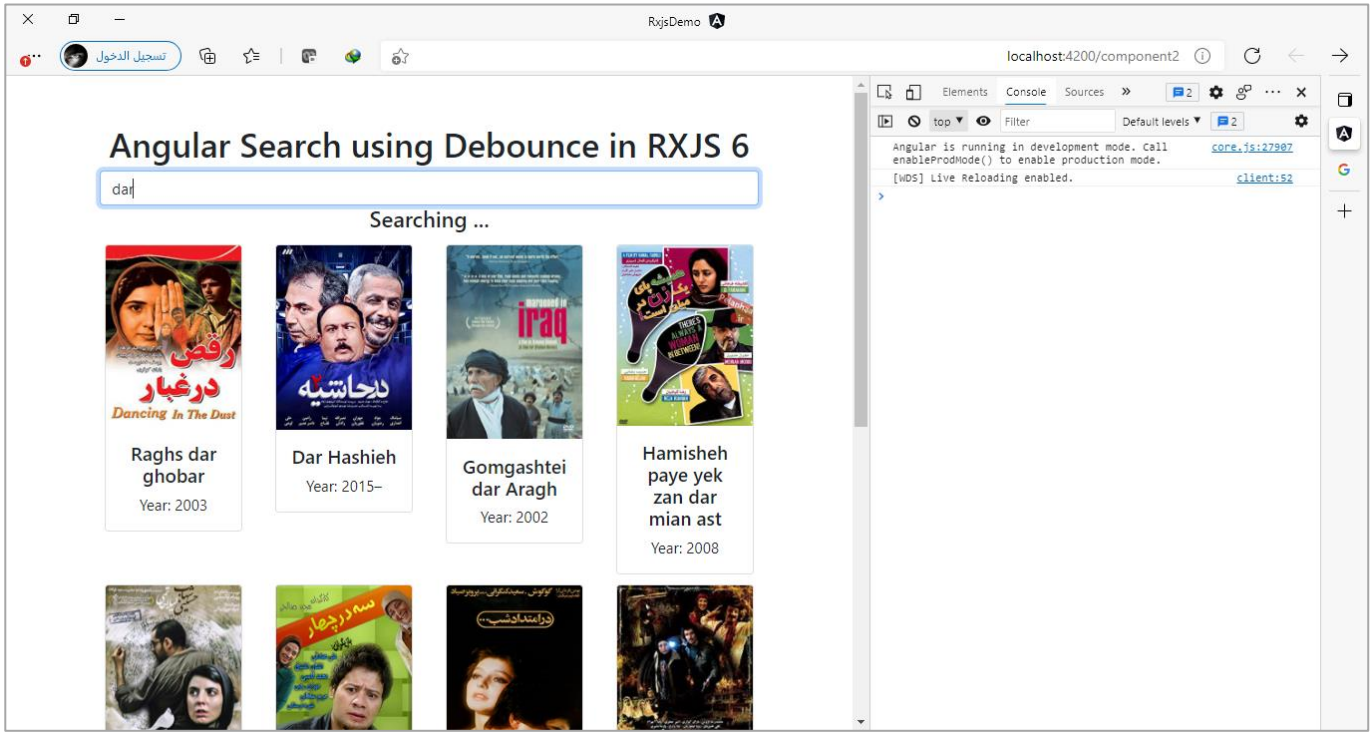
نلاحظ اننا استخدمنا عدد من الدوال وهذه الدوال تعلمنا أغلبها مسبقاً ما عدا الدالة tap والتي سوف نشرحها مسبقاً، وأول دالة هي map والفائدة من استخدامنا لها هو استخراج القيمة المدخلة في مربع النص، ومن ثم قمنا بفلترة هذه القيم بتمريرها على الدالة filter والسبب لكي نبحث ونتصل بالAPI في حالة كانت القيم اكثر من حرفين اما إذا كانت حرفين وأقل فنتجاهل القيم ولا نفعل شيء، ومن ثم استخدمنا الدالة tap لكي نضع قيمة المتغير isSearching تساوي true وهذا المتغير نستخدمه لكي نظهر أو نُخفي رسالة للمستخدم في ملف template تعلمه ان التطبيق يُجري عملية بحث، ومن ثم استخدمنا الدالة debounceTime ومررنا لها القيمة 1000 ميلي ثانية (ثانية واحدة) لكي لا نُرهق السيرفر بكثرة الطلبات لأنه في حال لم نستخدم هذه الدالة فإن أي تعديل او أي قيمة يتم كتابتها في مربع البحث سيتم ارسال طلب لها في السيرفر لذلك نستخدم هذه الدالة بحيث كل ثانية نأخذ القيم ونُرسلها لسيرفر وفي حال قام المستخدم بكتابة أي قيمة قبل انتهاء الثانية ستقوم هذه الدالة بإخذ القيم والانتظار وتصفير العداد وانتظار ثانية من جديد قبل ارسالها إلى السيرفر وبذلك قللنا من كثرة الاتصال بالسيرفر، ومن ثم استخدمنا الدالة distinctUntilChanged لكيلا نمرر نفس القيم وانما نسمح بالقيم المختلفة، ومن ثم استخدمنا الدالة switchMap لسببين الأول اننا سنتعامل مع Higher Order Observables ويتضح هذا جلياً في اتصالنا بالسيرفر باستخدام الدالة ajax لأن هذه الدالة تُعيد Observable اما السبب الثاني لأننا نُريد ان نُلغي الاتصال السابق في حال وجوده وابداله بالاتصال الجديد بالسيرفر أي بمعنى في حال كان هنالك Observable موجود قم بإلغائه وضع مكانه الObservable الجديد (الرجاء الرجوع إلى الجزء الخاص بالswitchMap لمعلومات أكثر)، ومن ثم استخدمنا الدالة map لكي نستخرج فقط الخاصية response من الاستجابة الخاصة بالسيرفر وطبعاً قد تختلف اسم هذه الخاصية من سيرفر إلى آخر ومن خاصية إلى أخرى، وأخيرا عملنا subscribe وعرضنا القيم.





نلاحظ في حالة وجود حرفين فقط فلن يحدث شيء وهذا هو المتوقع لأننا استخدمنا الدالة filter ومررنا لها الشرط الذي تكلمن عنه سابقاً.





نلاحظ عند رجوع الاستجابة من السيرفر قمنا بعرضها.

وفي الحقيقة هذه هي استخدامات دالة debounceTime والدالة distinctUntilChanged في عالم Angular.

auditTime() -2-6-2-3:

هذه الدالة مشابهة لدالة debounceTime والفرق الوحيد ان هذه الدالة لا تُعيد تصفير الوقت عندما يتم عمل emit لقيمة جديدة اثناء فترة الانتظار، وهي بذلك مشابهة لعمل البواب الذي يفتح البوابة كُل فترة زمنية محددة لدخول الطلاب وبعد دخولهم جميعاً يرجع ينتظر فترة زمنية مشابهة لمقدار الفترة الزمنية التي انتظرها سابقاً ومن ثم يقوم بفتح البوابة سواء كان هنالك طلاب ام لم يكن.

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { interval } from 'rxjs';
import { auditTime, tap} from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  form: FormGroup = new FormGroup({
    name: new FormControl(),
  });

  counter = 1;
```

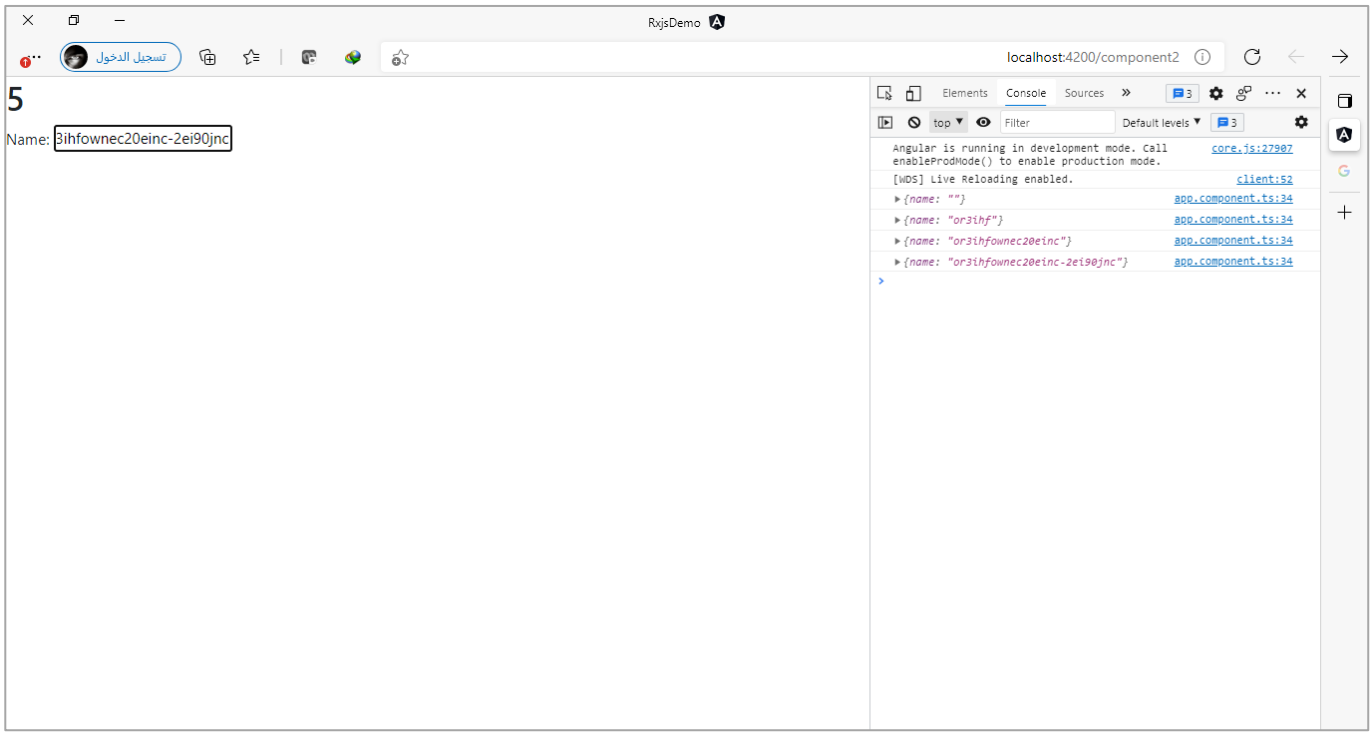
```
  constructor() {}

  ngOnInit(): void {
    interval(1000)
      .pipe(
        tap(() => {
          this.counter++;
          if (this.counter > 5) {
            this.counter = 1;
          }
        })
      )
      .subscribe();

    this.form.valueChanges
      .pipe(auditTime(3000))
      .subscribe((data) => console.log(data));
  }
}
```

كما نلاحظ سنُعيد تكرار نفس المثال السابق ولكن باستخدام هذه الدالة.

ملف app.component.html

```
<div>
  <h1>{{ counter }}</h1>
</div>
<form [formGroup]="form">Name: <input formControlName="name" /></form>
```

والنتيجة تكون كالتالي:



سوف تلاحظ عزيزي المتعلم ان هذه الدالة تنتظر اربع ثواني ومن ثم تعمل emit للقيم التي وصلتها ولا تهتم بوقت وصول هذه القيم إليها فقط الذي تهتم فيه هو انتظار فترة زمنية معينة ومن ثم تُرسل القيم التي لديها جميعها وترجع مرة أخرى تنتظر فترة زمنية مشابهة لسابقة وايضاً عند انتهاء الفترة الزمنية تُرسل القيم التي وصلتها وهكذا.

3-6-2-3- throttleTime():

سلوك هذه الدالة تختلف قليلاً عن الدوال السابقة، حيث تقوم بقراءة اول قيمة مباشرة ومن ثم تنتظر فترة زمنية معينة يتم تمريرها لها ومن ثم بعد انتهاء هذه الفترة تعود لقراءة القيم المتوفرة، مع انها قد تتجاهل القيم التي تم عمل emit لها اثناء فترة الانتظار.

ومن أشهر استخداماتها هو تعاملها مع احداث Events مؤشر الفأرة، حيث في بعض الأحيان لا نُريد ان يتم قراءة كُل ضغطة تتم على الزر من قبل المستخدم وخصوصاً إذا كانت بشكل مُتوالي ومتكرر وانما نقرأ اول قيمة (نقرة) ومن ثم ننتظر فترة زمنية معينة مع تجاهل أي ضغطات من قبل المستخدم تتم على الزر اثناء هذه الفترة الزمنية وعند انتهاء الفترة ولم تحدث استجابة نُعيد قراءة قيمة (نقرة) أخرى، ومن أمثلتها زر التصويت حيث عندما يقوم المستخدم بالنقر على الزر سيقوم التطبيق بالاتصال بالسيرفر مرة واحدة فقط وارسال القيمة بدلاً من تكرار ارسال القيم متوالية.

ولتوضيح لنُعطي مثال حيث نوضح عن طريقه كيفية الاستفادة من هذه الدالة وذلك عن طريق وجود زر وعند الضغط على هذا الزر يتم الاتصال بقاعدة بيانات عن طريق api معين لجلب مجموعة من البيانات، لذلك سوف نستخدم هذه الدالة بحيث إذا تم الضغط على هذا الزر نأخذ اول ضغطة ومن ثم نُعطل الزر وعند رجوع الاستجابة من السيرفر نُعيد تفعيل هذا الزر مرة أخرى، والذي نُريد ان نوضحه هنا انه يمكن الاستفادة من هذه الدالة لأن من خصائصها انها تأخذ اول قيمة (نقرة) لذلك نستفيد من هذه الميزة لتحقيق ما نصبو اليه في هذا المثال، كالتالي:

ملف app.component.html

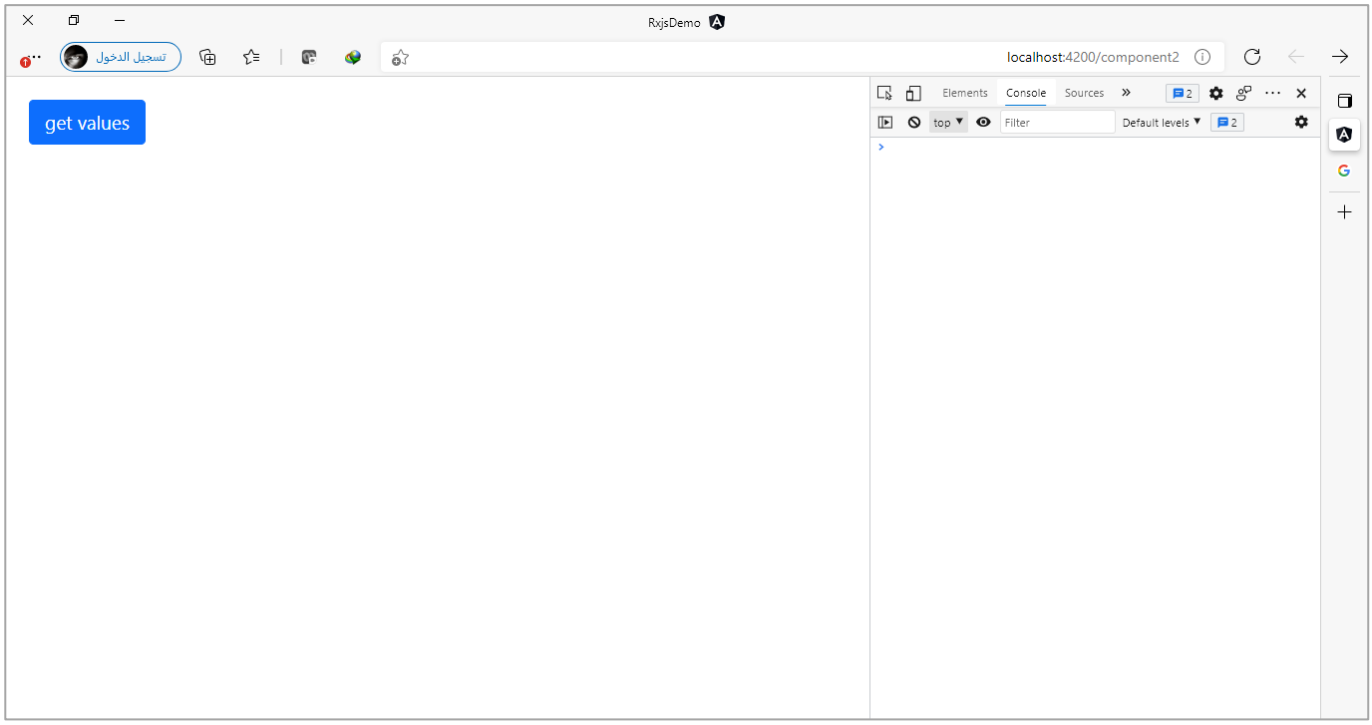
```
<button #btn class="btn btn-primary btn-lg m-4">get values</button>
```

ملف app.component.ts

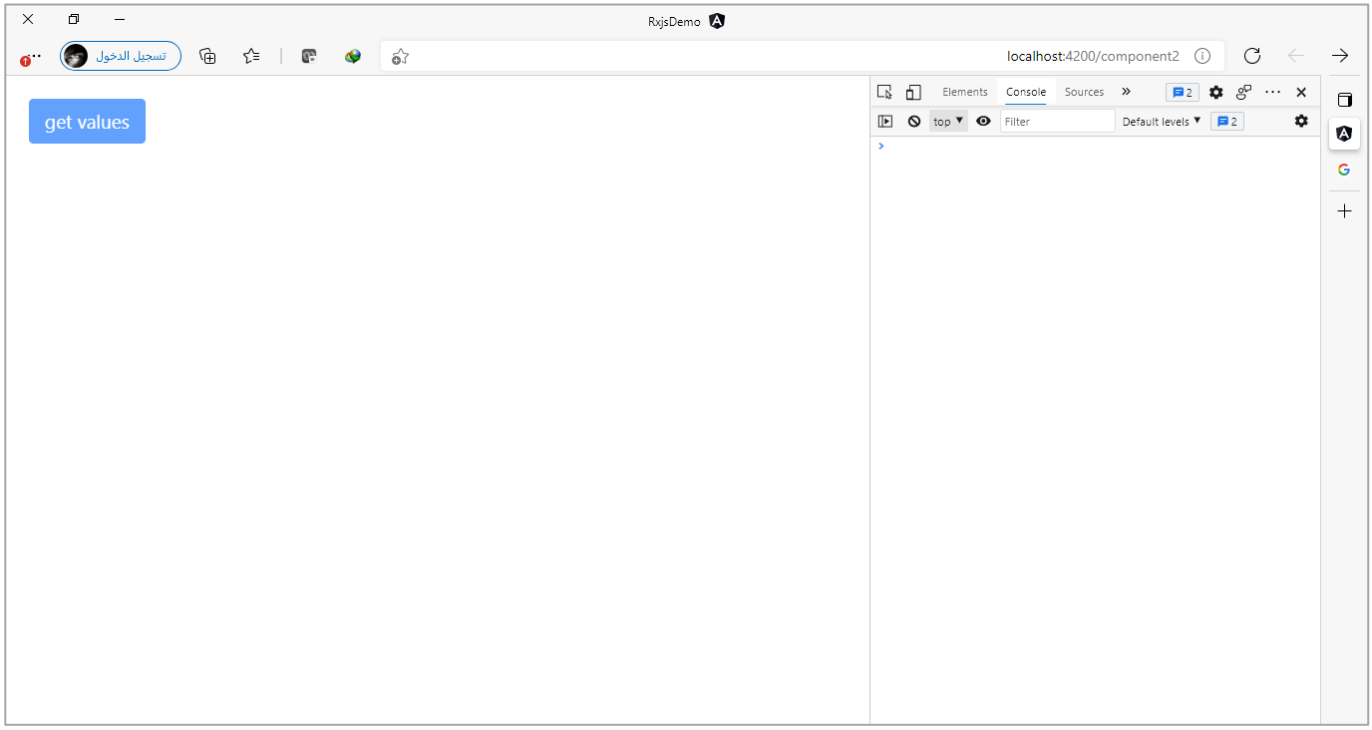
```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { delay, mergeMap, tap, throttleTime } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  @ViewChild('btn', { static: true })
  private btn: ElementRef<HTMLButtonElement>;
  constructor() {}

  ngOnInit(): void {
```

```
    fromEvent(this.btn.nativeElement, 'click')
      .pipe(
        throttleTime(1000),
        tap(() => (this.btn.nativeElement.disabled = true)),
        delay(2000),
        mergeMap(() => {
          return ajax.getJSON('https://jsonplaceholder.typicode.com/posts');
        }),
        delay(2000),
      )
      .subscribe((response: any[]) => {
        console.log(response);
        this.btn.nativeElement.disabled = false;
      });
  }
}
```

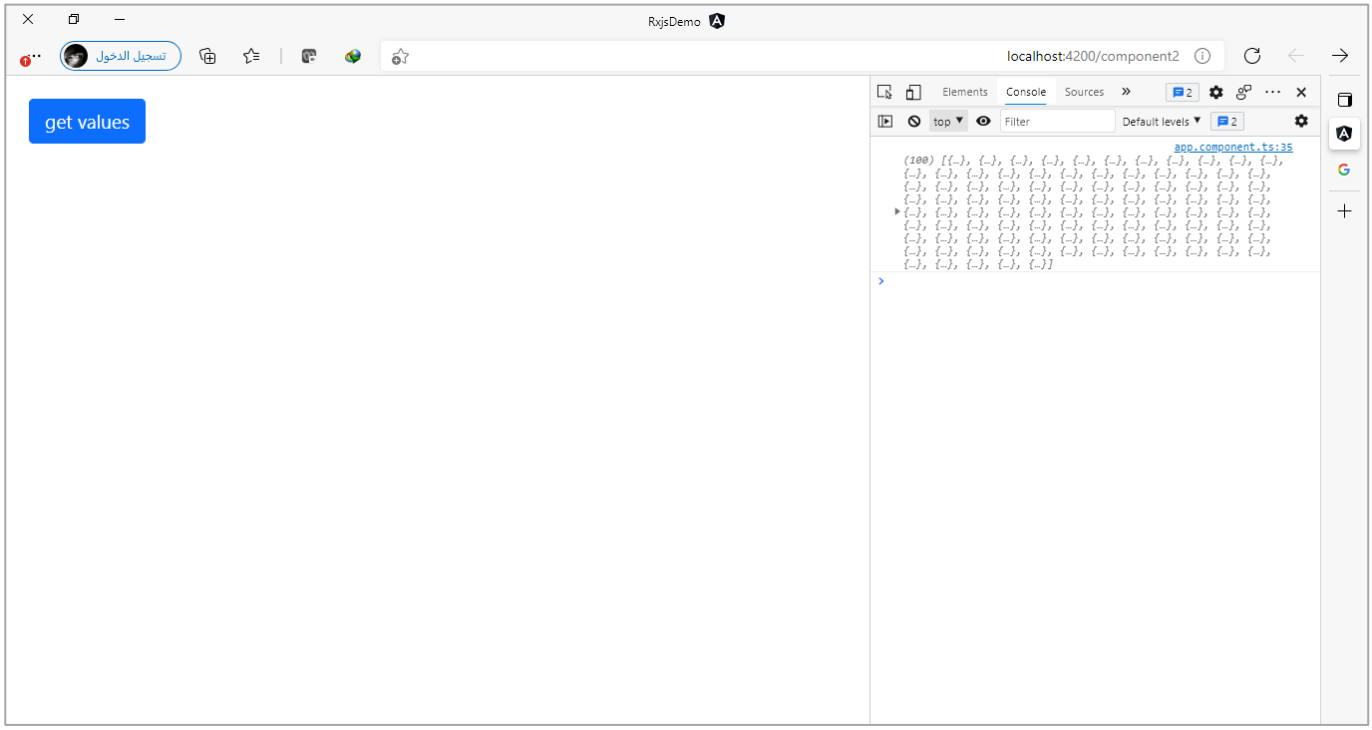
نلاحظ استخدمنا الدالة delay لكي نُحاكي تأخير الاستجابة من السيرفر لمدة ثانيتين وسوف نتكلم عنها بوقت لاحق بإذن الله، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



ولنقم الآن بالضغط على الزر ونرى النتيجة:



نلاحظ تم تعطيل الزر مع استقبال اول نقرة وتنفيذ الاتصال بالسيرفر.



واخيراً تمت الاستجابة وعرض البيانات بنجاح في console مع إعادة تفعيل الزر.

3-2-7- Operators first()/last():

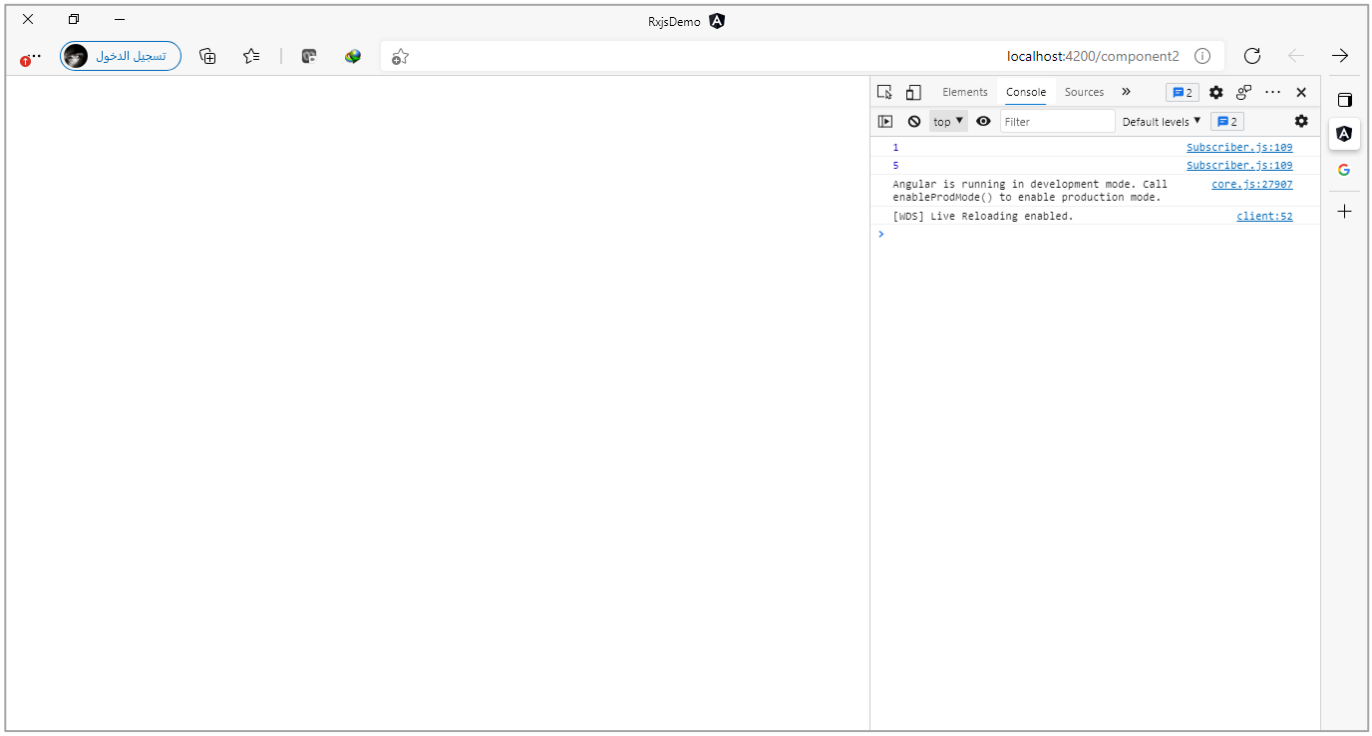
هاتين الدالتين تأخذان اول قيمة او آخر قيمة من كل Observable، كما في المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
```

```
import { first, last } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    const firstOpt = of(1, 2, 3, 4, 5, 6).pipe(first());
    const lastOpt = of(1, 2, 3, 4, 5).pipe(last());

    firstOpt.subscribe(console.log);
    lastOpt.subscribe(console.log);
  }
}
```

والنتيجة:



كما نستطيع تمرير بارامتر اختياري على شكل دالة Function ونستطيع عن طريقه تحديد مثلاً قيمة محددة نُريد اقتطاعها من الObservable، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { first, last } from 'rxjs/operators';

@Component({
```

```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    const firstOpt = of(1, 2, 3, 4, 5, 6).pipe(first((val) => val === 3));
    const lastOpt = of(1, 2, 3, 4, 5).pipe(last((val) => val > 3));

    firstOpt.subscribe(console.log);
    lastOpt.subscribe(console.log);
  }
}
```

والنتيجة:

واخيراً نستطيع تمرير قيمة افتراضية في حالة القيم المراد اقتطاعها غير موجودة في الObservable، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { first, last } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    const firstOpt = of(1, 2, 3, 4, 5, 6).pipe(
      first((val) => val > 6, 'Nothing from First Operators')
    );
    const lastOpt = of(1, 2, 3, 4, 5).pipe(
      last((val) => val === 0, 'Nothing from Last Operators')
    );
    firstOpt.subscribe(console.log);
    lastOpt.subscribe(console.log);
  }
```

```
}
```

والنتيجة:

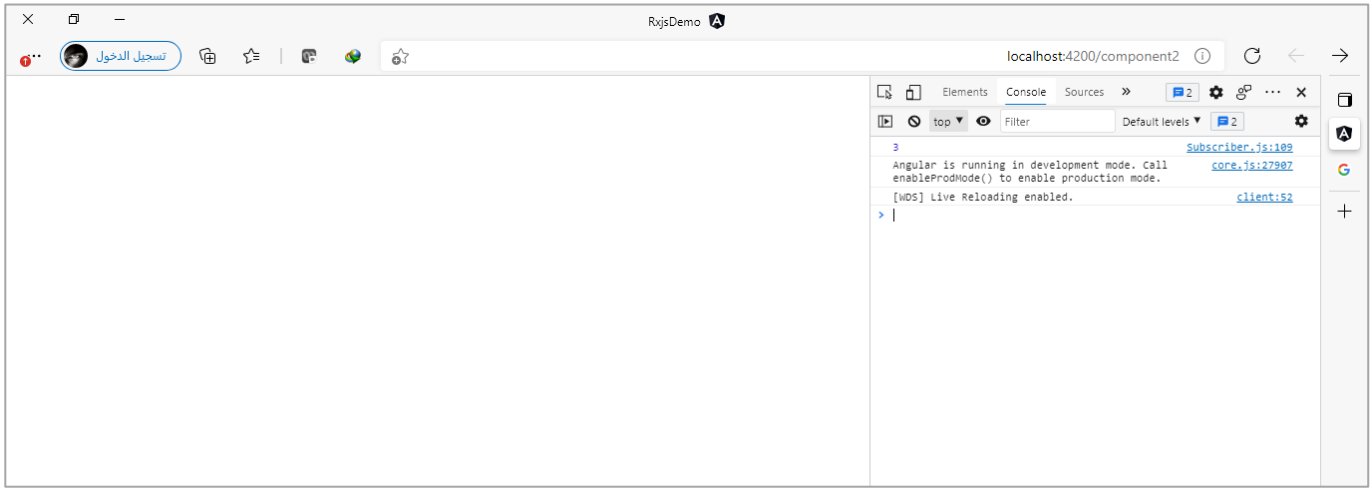
3-2-8- Operator single():

هذه الدالة مشابهة لدالتي last وfirst في حالة تمرير دالة Function لهما بغرض استقطاع قيمة معينة وفق شرط معين، والفرق ان في هذه الدالة اجباري اما في الدالتين فهو اختياري، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { single } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {
    of(1, 2, 3, 4, 5, 6)
      .pipe(single((val) => val === 3))
      .subscribe(console.log);
  }
}
```

والنتيجة:



Join Operators -3-3:

وتسمى ايضاً combination operators، وهي مشابهة لعملها مع دوال Join التي تم التطرق لها سابقاً في قسم Static Operators، وفي الحقيقة هي مشابهة لها حتى بالأسماء إلى الإصدار السابع من مكتبة Rxjs تم تغيير الأسماء لتفريق بينها فقط بالتسمية بإن هذه الدوال هي Pipeable Operators وليست Static Operators اما الوظيفة فهي واحدة.

Static Operators	Pipeable Operators
zip	zipWith
concat	concatWith
merge	mergeWith
race	raceWith
combineLatest	combineLatestWith

كما نلاحظ تم إضافة الكلمة with لكل دالة لتفريق ومعرفة ان هذه الدوال تنتمي إلى Pipeable Operators، والسؤال الذي قد يتراود إلى ذهنك عزيزي المتعلم هو ما الفائدة منها إذا دوال Static Operators موجودة، والسبب انه في بعض الأحيان الObservable الثاني الذي نُريد ان نعمل له دمج يكون فرعي من الObservable الأول او انه يعتمد عليه لجلب قيمه، كأن يكون لدينا بيانات معلمين ونُريد ان نعمل دمج مع بيانات الطلاب لكل معلم ففي هذه الحالة الObservable الذي يحتوي على قيم بيانات الطلاب يعتمد على الObservable الذي يحتوي على بيانات المعلمين أي اننا نجلب بيانات المعلمين وبناءً على id الخاص بكل معلم نجلب بيانات الطلاب لذلك في مثل هذه الحالة نحتاج في حالة اردنا ان ندمج هذه الObservables مع بعضها البعض فلابد ان نستخدم أحدى هذه الدوال مع إضافة with لها.

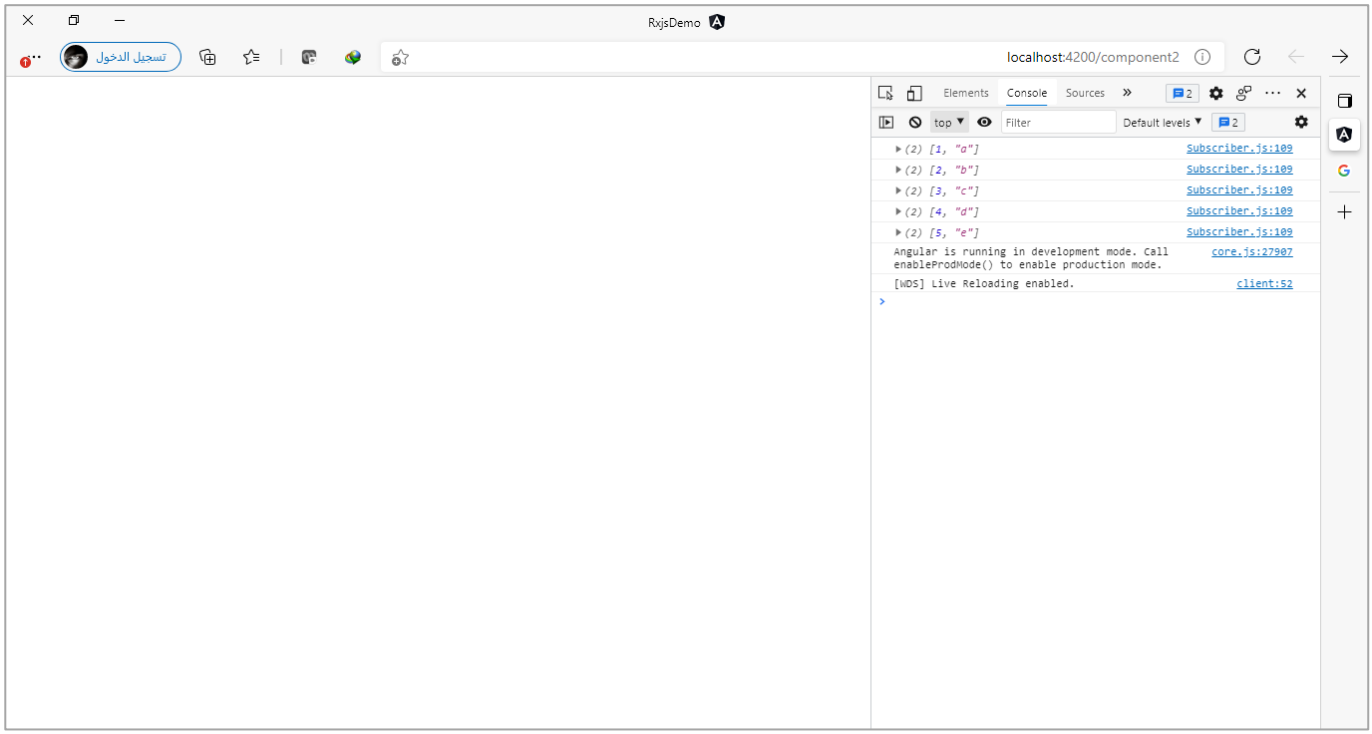
ومنعاً لتكرار فلن أعيد شرح هذه الدوال مرة أخرى وذلك بسبب انها متشابهة بكل شيء فلذلك نكتفي بما ذكرناه سابقاً.

واما من ناحية طريقة استخدام هذه الدوال فسوف اضرب مثال على الدالة zip مع العلم ان باقي الدوال نتعامل معها بنفس الطريقة، والتي هي كتابة الObservable الأساسي ومن ثم نستخدم pipe وفيها نستخدم احدى دوال الدمج السابقة لندمج معها الObservable الثاني او أي عدد نُريده من الObservables.

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { zipWith } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  ngOnInit(): void {
    of(1, 2, 3, 4, 5, 6, 7, 8)
      .pipe(zipWith(of('a', 'b', 'c', 'd', 'e')))
      .subscribe(console.log);
  }
  constructor() {}
}
```

نلاحظ الObservable الأول يحتوي على قيم رقمية وقمنا بدمجه مع الObservable الثاني الذي يحتوي على قيم نصية، وسوف تكون النتيجة، كالتالي:



كما تلاحظ تم دمج كلا الObservables مع بعضها البعض وليس هذا فحسب وانما تم تطبيق كافة مفاهيم الدالة zip من خلال الدالة zipWith لذلك قلت سابقاً انني لن اشرح هذه الدوال منعاً لتكرار.

ولكن هنالك في الحقيقة ثلاث دوال، غير الموجودة في الجدول السابق وتعتبر جديدة لذلك سوف استعرضها بالشرح والتفصيل.

withLatestFrom() -1-3-3:

تعتبر هذه الدالة شائعة الاستخدام، وتقوم فكرتها في حال كان لدينا Observable رئيسي وفي حال تغير قيم هذا Observable يتم دمج هذه القيمة مع Observable آخر. ويجب الانتباه انها لا تُراقب التعديلات في Observable الثاني وانما تُراقب فقط Observable الأول.

ولتوضيح لنعطي مثال، يكون فيه قائمتين، قائمة تحتوي على بيانات اقسام وقائمة أخرى تحتوي على بيانات الموظفين، ونُريد انه في حالة اختيار الموظفين نقوم بدمج هؤلاء الموظفين المحددين مع القسم الخاص بهم من قائمة الأقسام.

ولتسهيل سوف استخدم قائمة جاهزة تُقدم إمكانيات جيدة مع سهولة الاستخدام، وتستطيع الاطلاع على جميع مميزاتها وتفاصيلها على هذا الرابط: ng-multiselect-dropdown - npm (npmjs.com) لذلك لنذهب إلى مشروع Angular ونذهب إلى terminal ونتأكد بأننا على نفس مسار المشروع ونكتب الأمر التالي: `npm i ng-multiselect-dropdown`، ولمعرفة كيفية التعامل مع terminal واضافة المكتبات الخارجية في مشاريع Angular الرجاء مراجعة كتابي Angular Environment Setup مع العلم انه جزء من هذه السلسلة.

وبعد إضافة هذه المكتبة نذهب إلى ملف app.module.ts ونضيف السطر التالي:

ملف app.module.ts

```
import { HttpClientModule } from '@angular/common/http';
import { NgModule } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NgMultiSelectDropDownModule } from 'ng-multiselect-dropdown';


@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    HttpClientModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    NgMultiSelectDropDownModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

والآن بعدما اضفنا المكتبة وقمنا بتهيئتها، لذهب الآن إلى ملف app.component.html ونُضيف الMarkup التالي:

ملف app.component.html

```
<div>
  <ng-multiselect-dropdown
    [placeholder]="'Please Select Department ...'"
    [settings]="{ singleSelection: true, enableCheckAll: false }"
    [data]="departments"
    [(ngModel)]="selectedDepartment"
    (onSelect)="onDepartmentSelect()"
    (onDeSelect)="onDepartmentSelect()"
    class="m-3"
  >
  </ng-multiselect-dropdown>
  <ng-multiselect-dropdown
    [placeholder]="'Please Select Employees ...'"
    [settings]="{ singleSelection: false, enableCheckAll: false }"
    [data]="employees"
    [(ngModel)]="selectedEmployees"
    (onSelect)="onEmployeeSelect()"
    (onDeSelect)="onEmployeeSelect()"
    class="m-3"
  >
  </ng-multiselect-dropdown>
</div>
```

وفي ملف style نُضيف التنسيقات التالية:

ملف app.component.scss

```
div {
  display: flex;
  flex-direction: column;
  justify-content: center;
}
```

وأخيراً في ملف class نضيف الاسطر البرمجية التالية:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of, Subject } from 'rxjs';
import { map, withLatestFrom } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  employees = ['Faisal', 'Fahd', 'Saad', 'Bader', 'Ali'];
  departments = [
```

```
    'Production',
    'Human Resource Management',
    'Accounting and Finance',
  ];


  selectedEmployees = [];
  selectedDepartment = '';
  subjectEmployees = new Subject<string[]>();
  subjectDepartments = new Subject<string>();

  ngOnInit(): void {
    this.subjectEmployees
      .pipe(
        withLatestFrom(this.subjectDepartments),
        map(([emp, dept]) => {
          return { employees: emp, department: dept.toString() };
        })
      )
      .subscribe(console.log);
  }
  constructor() {}

  onDepartmentSelect(): void {
    this.subjectDepartments.next(this.selectedDepartment);
  }

  onEmployeeSelect(): void {
    this.subjectEmployees.next(this.selectedEmployees);
  }
}
```

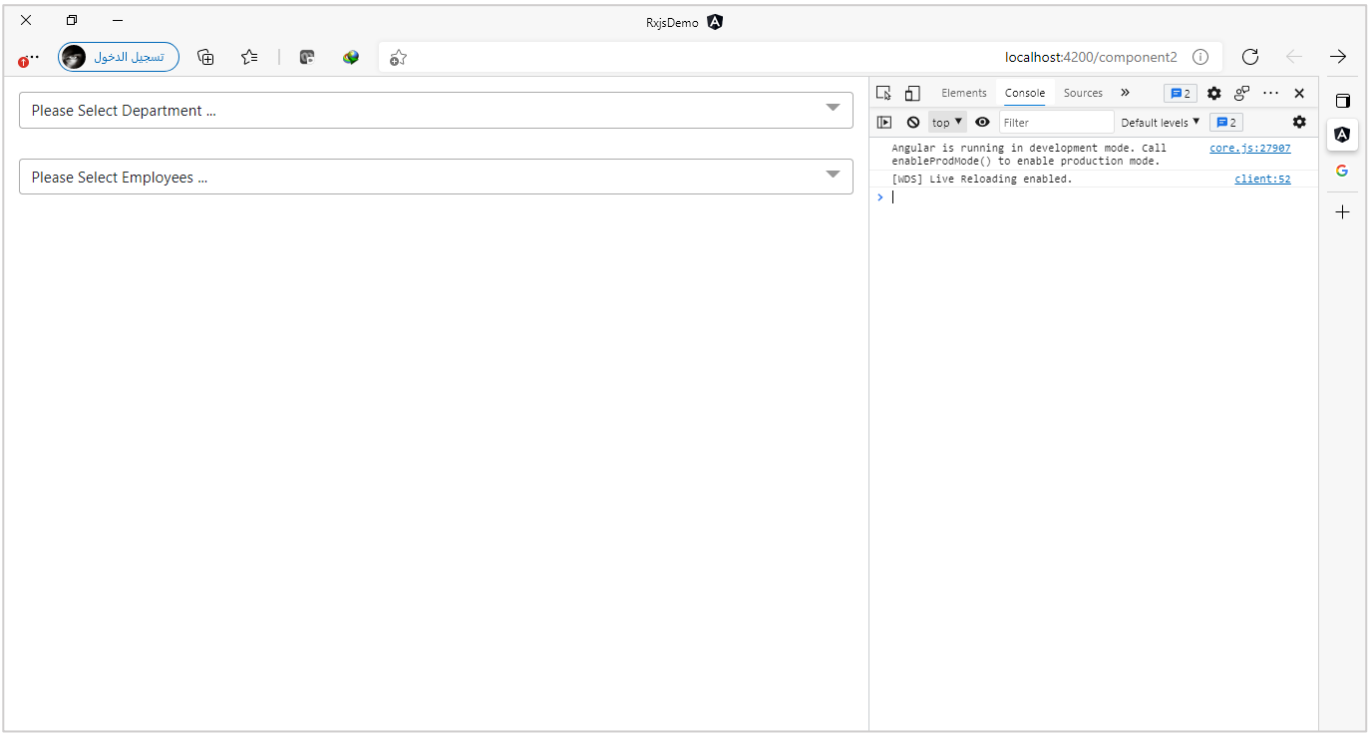
في النقطة 1 تم تعريف مجموعة من البيانات العشوائية لكي نقوم بملء القائمتين.

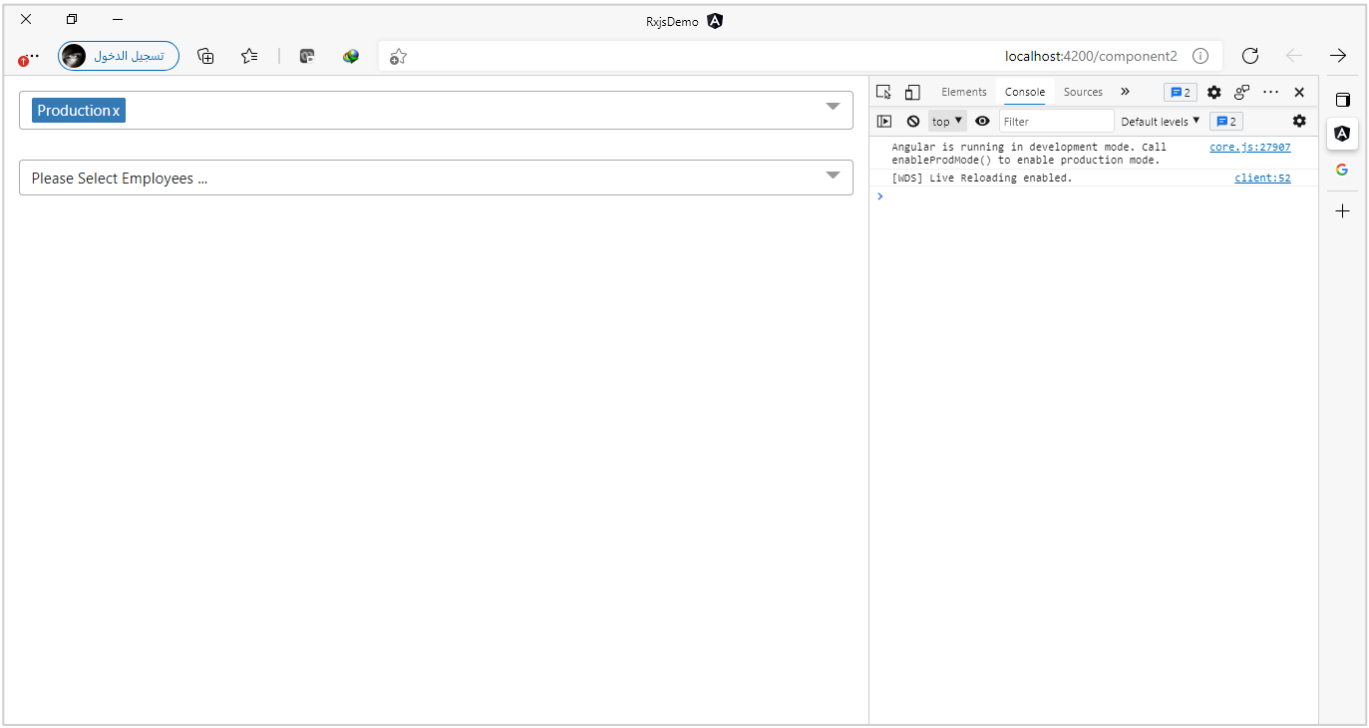
اما في النقطة 2 فقمنا بتعريف مجموعة من المتغيرات بحيث اول متغيرين يحملان القيم التي يتم اختيارها من القائمتين وبما انه يمكن اختيار اكثر من موظف فلذلك تم تعريفها على انها مصفوفة بعكس المتغير الثاني لأنه لا يمكن اختيار إلا قسم واحد، اما آخر متغيرين فيمثلان الObservable التي سوف نقوم بدمجهما لاحقاً.

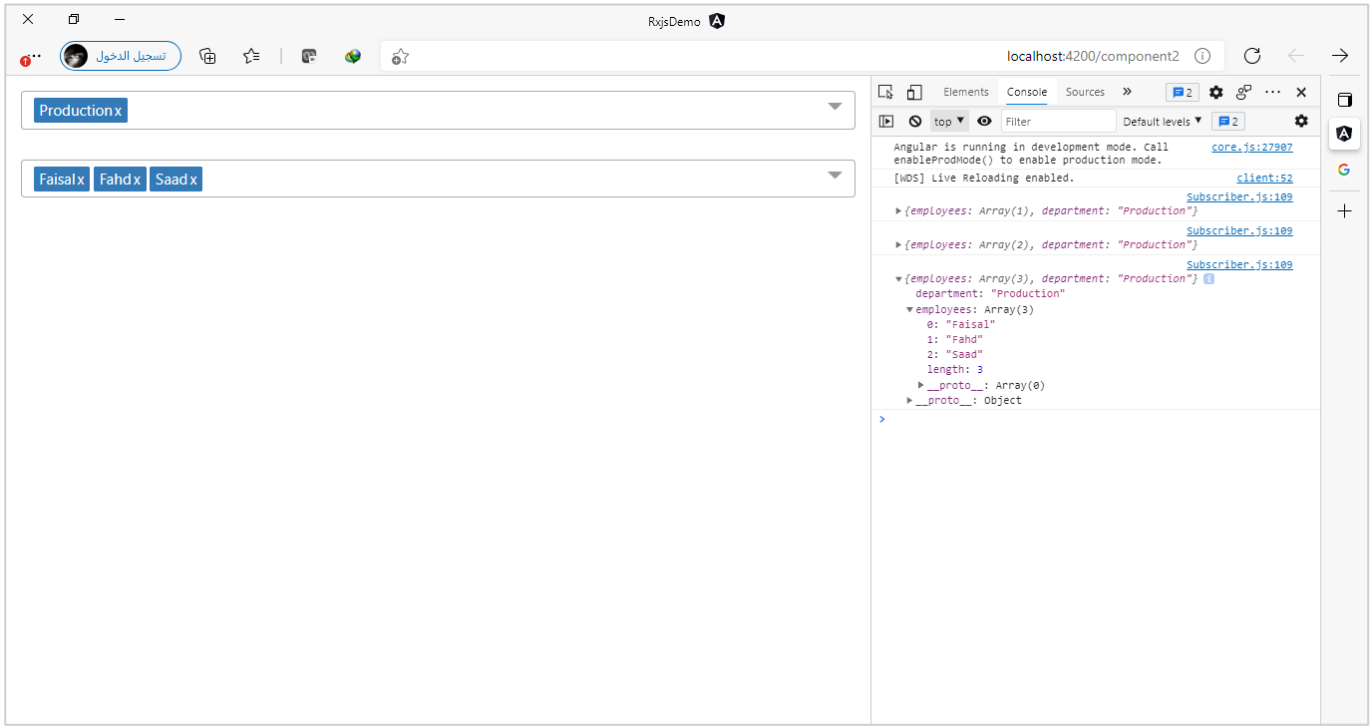
والنقطة 3 كتبنا الاسطر البرمجية اللازمة لدمج هذه الObservables مع بعضها البعض بحيث عند أي تغيير في قيم الObservable الخاص بالموظفين سيتم دمجها مع الObservable الخاص بالأقسام والعكس غير صحيح أي انه لا يتم الدمج في حالة أي تغيير يحدث في بيانات الأقسام. ومن ثم مررنا القيم على الدالة map لكي نُخرج البيانات بشكل منظم وهو على شكل كائن يحتوي على اثنين keys الأول باسم employees ويحتوي على مصفوفة من بيانات الموظفين اما الثاني باسم department فيحتوي على اسم القسم المختار بعد تحويله من مصفوفة إلى نص.

اما آخر نقطتين فالغرض منها قراءة القيم عندما يقوم المستخدم بالاختيار من القوائم.

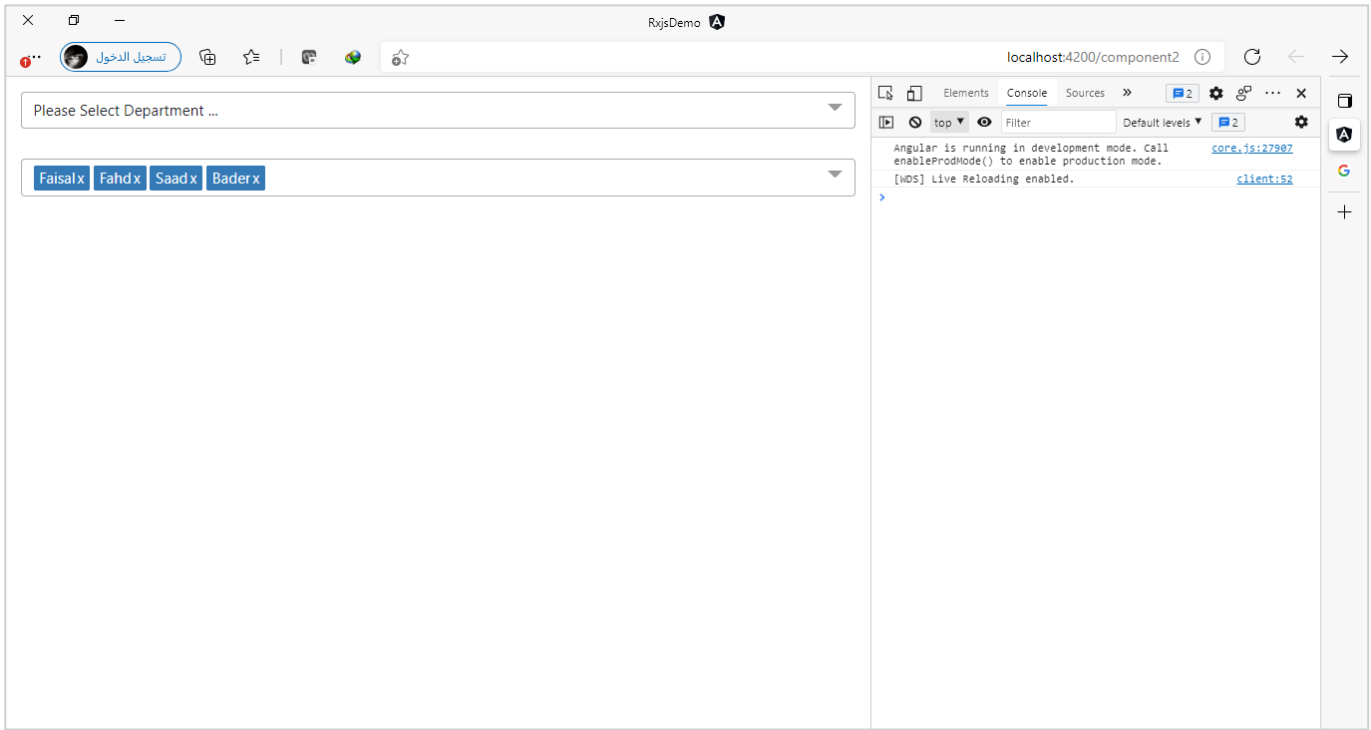
وبعد هذا الشرح المطول لشفرة البرمجة السابقة، لنشاهد النتيجة في المتصفح، كالتالي:







واخيراً تجدر الإشارة ان هذه الدالة لا تعمل في حال ان الObservable الثاني لم تبدأ قيمه، بعبارة أوضح في حال قمنا باختيار مجموعة من الموظفين ولكن لم نختار أي قيمة من قائمة الأقسام فإن هذه الدالة لن تعمل إلا ان يتم اختيار قيمة من قائمة الأقسام ومن ثم الرجوع والتعديل في قائمة الموظفين فعنها تبدأ الدالة withLatestFrom عملها.



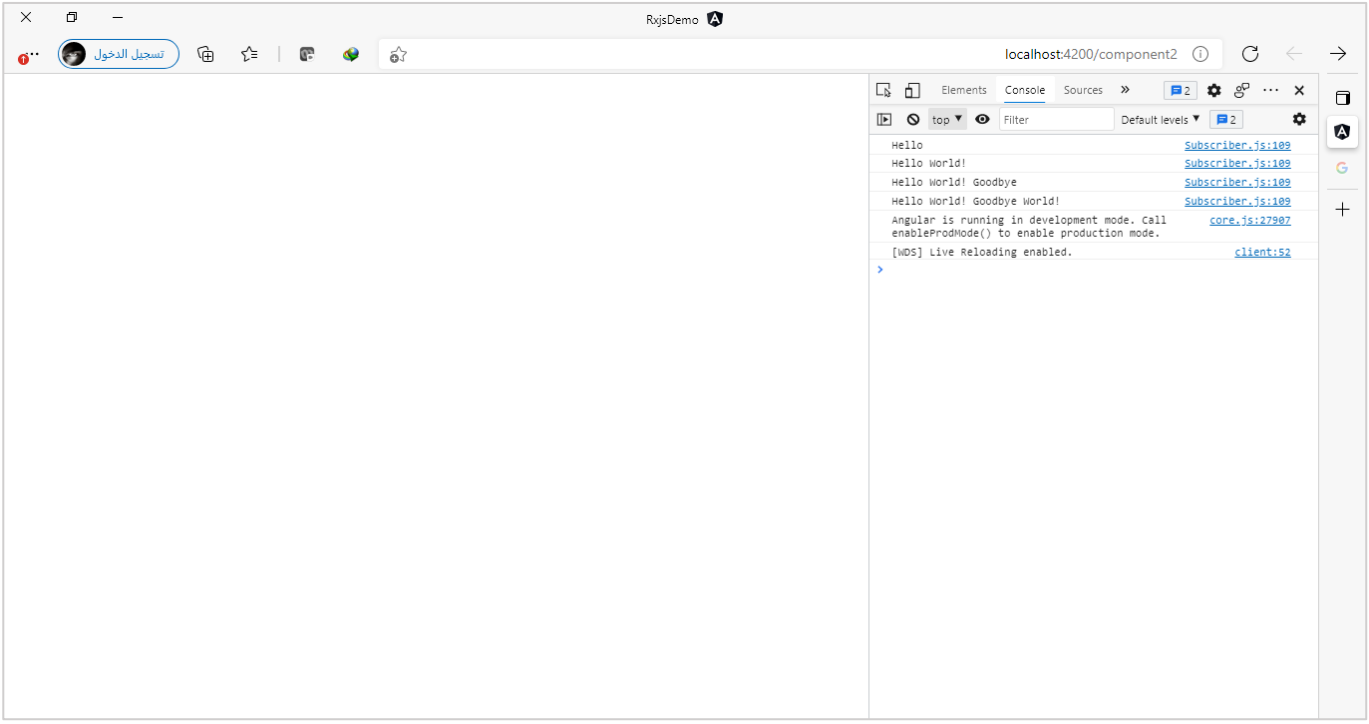
3-3-2- startWith():

تقوم هذه الدالة بعمل emit لقيمة قبل قيم الObservable الأساسي، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { scan, startWith } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    of('World!', 'Goodbye', 'World!')
      .pipe(
        startWith('Hello'),
        scan((acc, curr) => `${acc} ${curr}`)
      )
      .subscribe(console.log);
  }
  constructor() {}
}
```

والنتيجة:

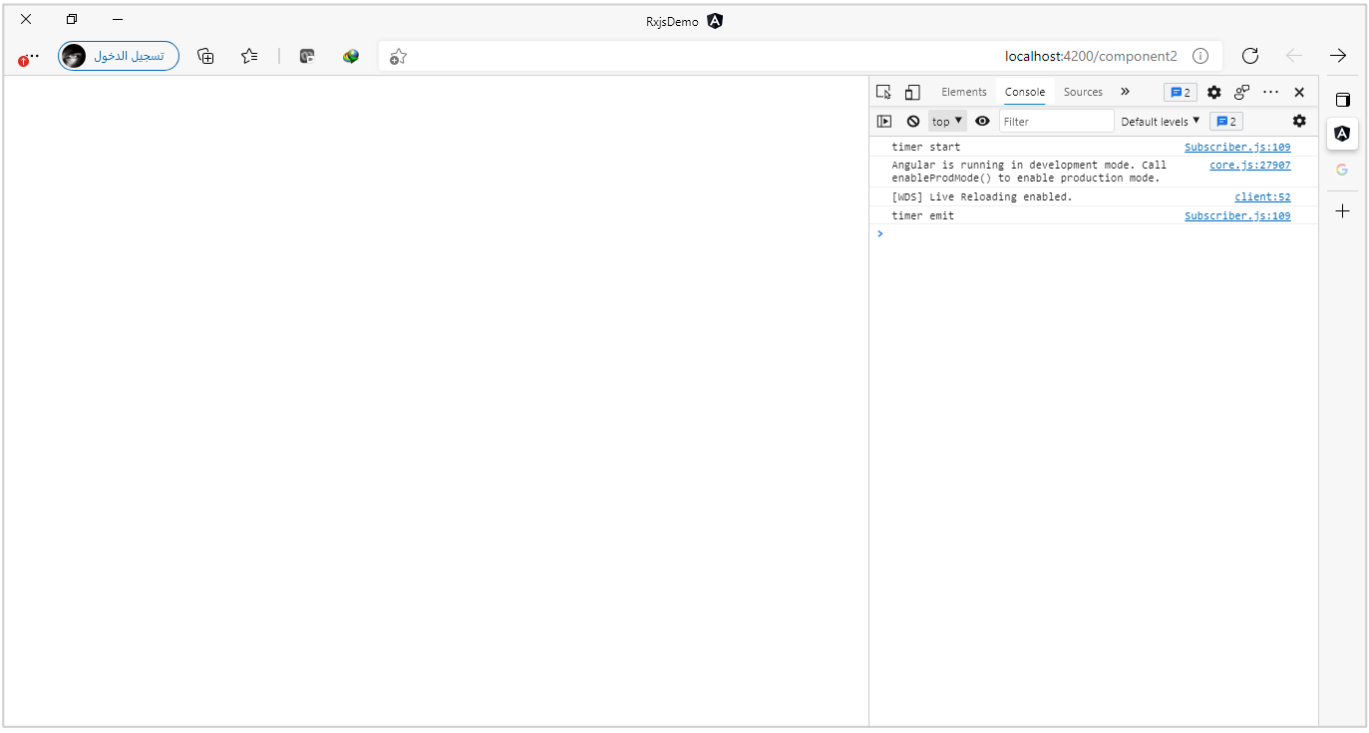


او المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { timer } from 'rxjs';
import { mapTo, startWith } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    timer(1000)
      .pipe(mapTo('timer emit'), startWith('timer start'))
      .subscribe(console.log);
  }
  constructor() {}
}
```

والنتيجة:



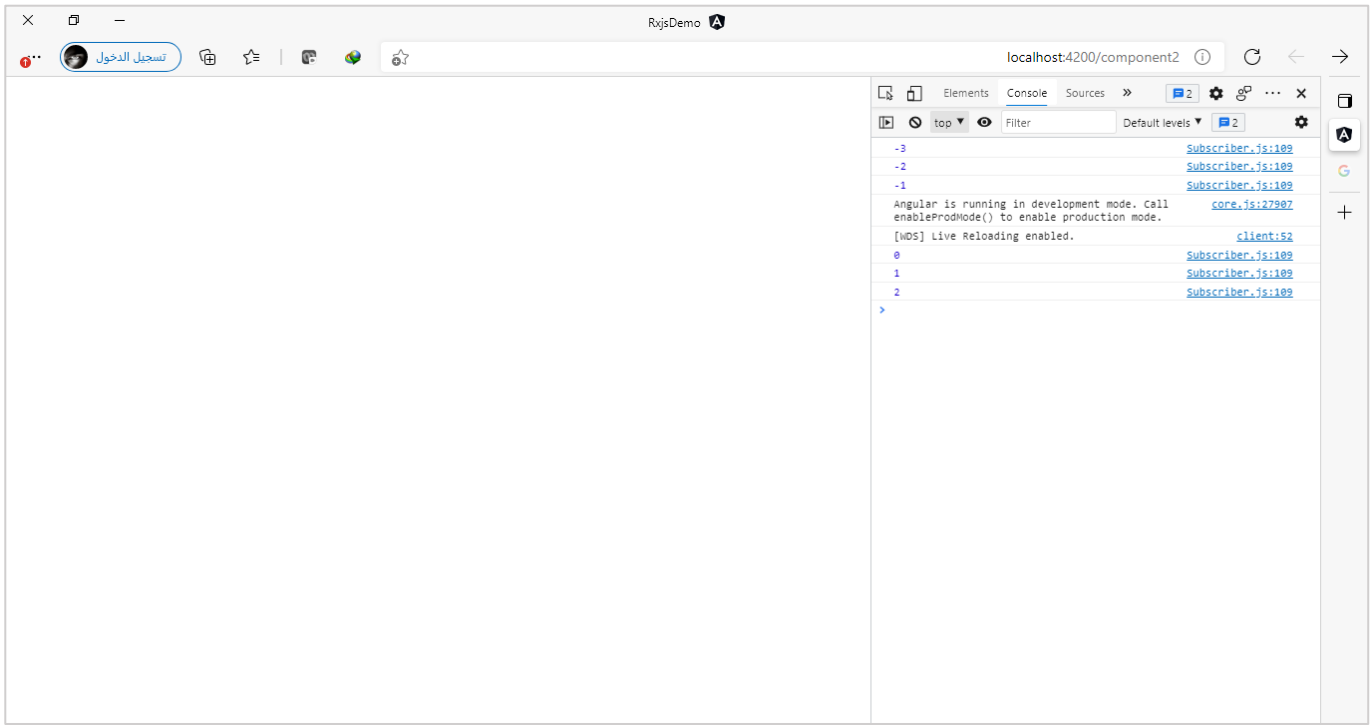
كما يمكن استخدامها لتمرير أكثر من قيمة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { startWith, take } from 'rxjs/operators';

@Component({
```

```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  ngOnInit(): void {
    interval(1000).pipe(take(3), startWith(-3, -2, -1)).subscribe(console.log);
  }
  constructor() {}
}
```

والنتيجة:



كما نستطيع استخدامها مع subject لإعطاء قيمة مبدئية في حال عدم الرغبة في استخدام BehaviorSubject.

3-3-3- endWith():

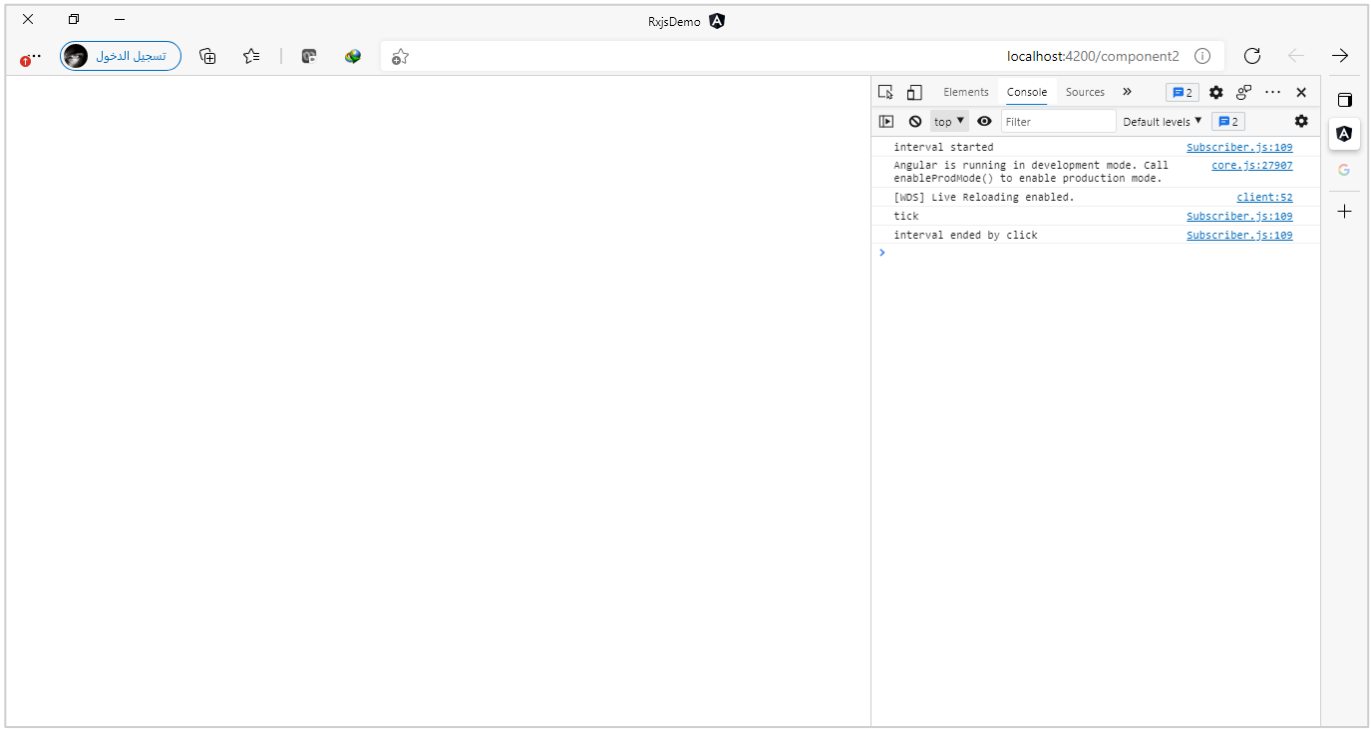
هذه الدالة عكس الدالة السابقة حيث تقوم بعمل emit بعد انتهاء الObservable الأساسي من عمل emit لجميع قيمه وقبل لا يعمل complete، ولتوضيح لنُعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { fromEvent, interval } from 'rxjs';
import { endWith, mapTo, startWith, takeUntil } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

```
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const documentClicks$ = fromEvent(document, 'click');
    interval(3000)
      .pipe(
        mapTo('tick'),
        startWith('interval started'),
        takeUntil(documentClicks$),
        endWith('interval ended by click')
      )
      .subscribe(console.log);
  }
  constructor() {}
}
```

والنتيجة:



4-3- Multicasting Operators:

هذه الدوال من اسمها تقوم بتحويل الObservable من Unicast إلى Multicast والتي تطرقنا إليها سابقاً في هذا الكتاب، لذلك لن أُعيد شرح هذه المفاهيم مرة أخرى وتستطيع الرجوع إلى القسم الخاص بالUnicast والMulticast لفهم أكثر عن هذه المفاهيم، وما يهمنا هنا هو الدوال التي ستقوم بتحويل من Unicast إلى Multicast وسوف نتطرق بإذن الله إلى دالتين هما share وshareReplay.

3-4-1- share() :

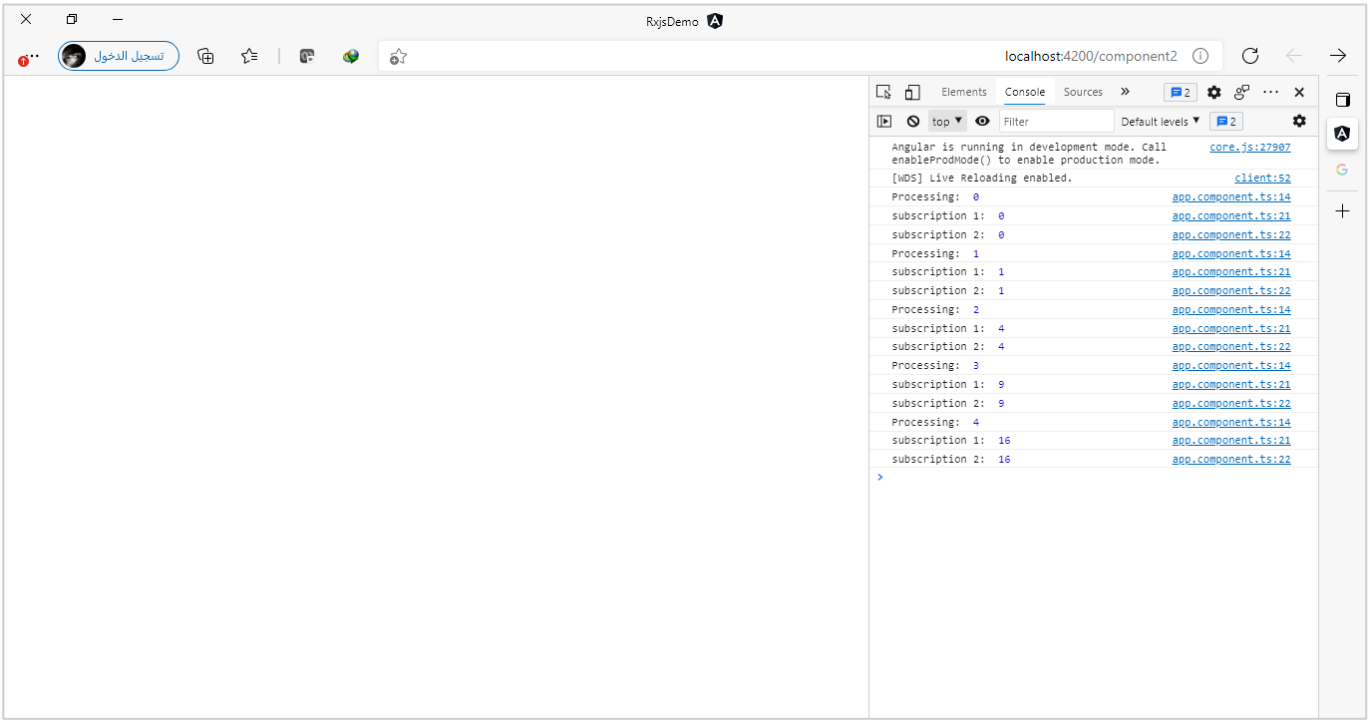
هذه الدالة لا تستقبل أي باراميتر وتقوم بعمل مشاركة لقيم الObservable مع جميع الSubscribers سواء كانت نفس القيم او تختلف، ولكن الفاصل الأساسي انها لا تقوم بإنشاء اتصال جديد بين الObserver والSubscriber عند كل Subscription جديد.

ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { map, share, take } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const source = interval(1000).pipe(
      map((x: number) => {
        console.log('Processing: ', x);
        return x * x;
      }),
      take(5),
      share()
    );

    source.subscribe((x) => console.log('subscription 1: ', x));
    source.subscribe((x) => console.log('subscription 2: ', x));
  }
  constructor() {}
}
```

سوف تلاحظ ان هذه السطر المؤشر عليه بالسهم في الأعلى سيتم استدعائه مره واحده فقط مع كل ارسال emit لقيمه جديدة لجميع Subscribers. وذلك بسبب اننا استخدمنا الدالة share، اما الtake فقد تكلمنا عنها سابقاً واستخدمتها هنا لكي آخذ فقط اول خمس قيم فالذي يهمنا ليس القيم وانما كيفية الاستفادة من الدالة share.



كما تلاحظ عزيزي المتعلم يتم طباعة Processing مرة واحد مع كل قيمة جديدة ومن ثم يتم عمل مشاركة share لكل Subscribers، وكما أشرت سابقاً وأعيد وأكرر هنا ليس المهم القيمة فالقيمة قد تتشابه لجميع الSubscribers ولكن ما يهمنا هو ان الاتصال واحد.

الآن حاول عزيزي المتعلم مع نفس ان تقوم بحذف الدالة share وشاهد النتيجة ومن ثم أعدها مرة أخرى وشاهد النتيجة ايضاً، وحاول المقارنة بين النتيجتين سواء باستخدام الدالة share او بدونها.

كما ايضاً لها استخدام مهم وهو تقليل الاتصال بالسيرفر في حال تم الاتصال بنفس المصدر source للبيانات، فمثلاً لو افترضنا اننا اتصلنا بالسيرفر بواسطة API معين واستقبلنا البيانات على شكل Observable في تطبيقنا، وبنفس الcomponent احتجنا ان نعمل اثنين Subscription لنفس هذه البيانات عن طريق async (وكما هو معروف ان هذه الدالة تعمل subscribe بشكل تلقائي للObservable)، كما في المثال التالي:

ملف app.component.ts

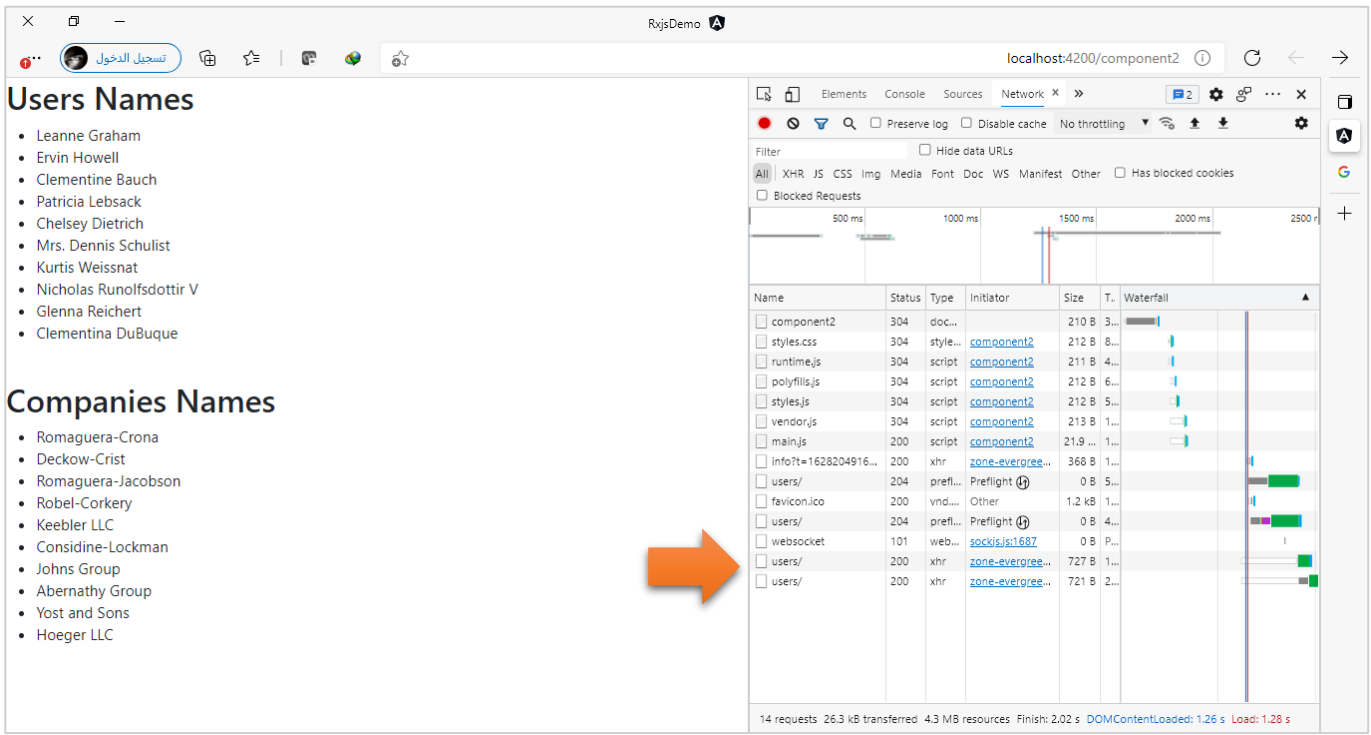
```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  source = ajax.getJSON('https://jsonplaceholder.typicode.com/users/');
  ngOnInit(): void {}
  constructor() {}
}
```

كما تلاحظ عزيزي المتعلم قمنا بالاتصال بالسيرفر عن طريق دالة ajax بواسطة API للاختبار فقط، وكما هو معروف ان دالة ajax تُعيد Observable لذلك استقبلناها في متغير اسميته source، وبذلك اصبح هذا المتغير يمثل الObservable ويحمل قيمه، والآن لنقوم بعمل اثنين Subscription لهذا المتغير في ملف template عن طريق الدالة async وليس في ملف class كما جرت العادة، كالتالي:

ملف app.component.html

```
<div>
  <h1>Users Names</h1>
  <ul>
    <li *ngFor="let source1 of source | async">{{ source1.name }}</li>
  </ul>
</div>
<br />
<div>
  <h1>Companies Names</h1>
  <ul>
    <li *ngFor="let source2 of source | async">{{ source2.company.name }}</li>
  </ul>
</div>
```

كما نلاحظ قمنا بعمل اثنين Subscription للمتغير source باستخدام الدالة async، بحيث الSubscription الأول لعرض أسماء الUsers في قائمة والثاني لعرض أسماء الشركات، مع العلم ان كلا هذين الSubscriptions مصدرهما Observable واحد والمتمثل بالمتغير source، الآن لنحفظ التعديلات ولنذهب إلى المتصفح ونفتح الconsole ومن ثم نذهب إلى التبويب network لكي نُشاهد الطلبات Request لسيرفر، كالتالي:

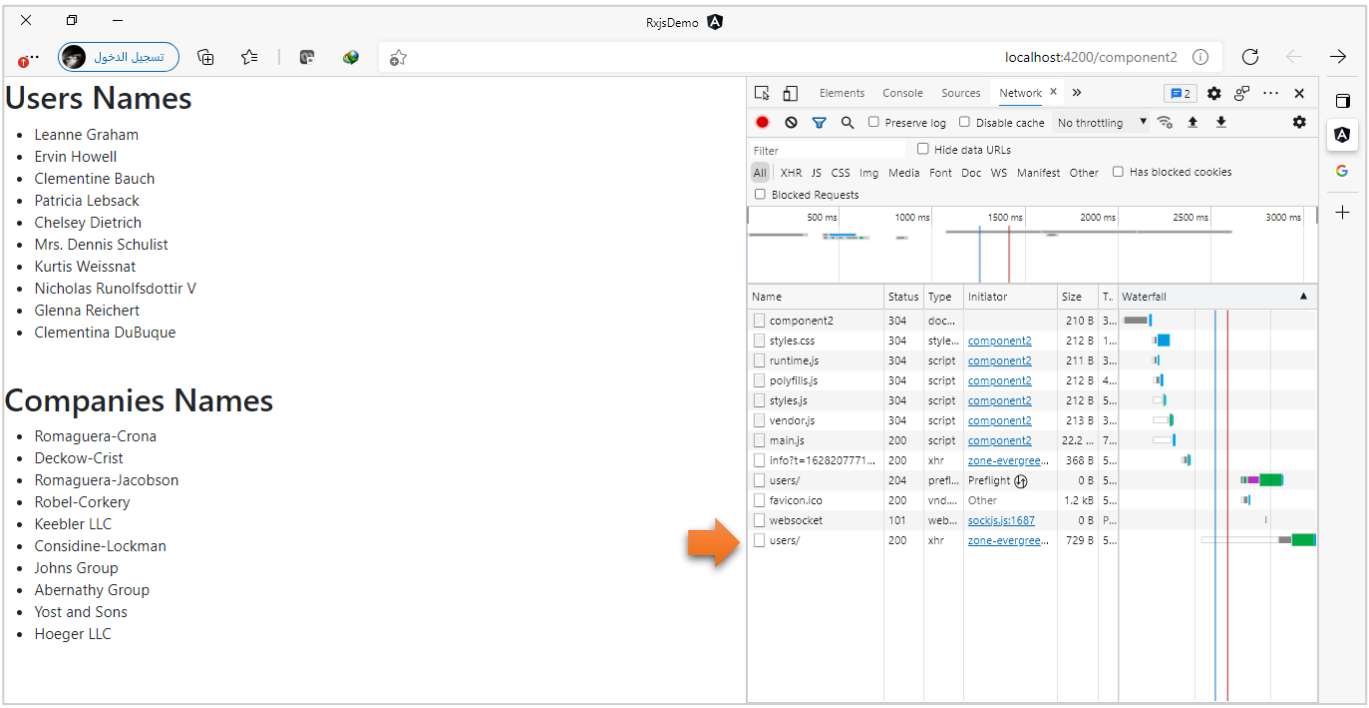


سوف تُشاهد عزيزي المتعلم ان هنالك اثنين طلب لسيرفر لجلب البيانات على الرغم من ان مصدر البيانات واحد، فلك ان تتخيل لو كان تطبيقك كبير ويستخدمه عدد هائل من المستخدمين بنفس الوقت ولك ان تتخيل كم عدد الطلبات والاستجابات التي تذهب او تأتي من السيرفر مما يرجع بالتأكيد على اداء تطبيقك من جهة واستهلاك موارد السيرفر من جهة أخرى، لذلك نستطيع التقليل من الطلبات باستخدام الدالة share حيث نعمل مشاركة لنفس المصدر Source لجميع الSubscribers بدون ان نُعيد الطلب من السيرفر لكل Subscriber، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { share } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  source = ajax
    .getJSON('https://jsonplaceholder.typicode.com/users/')
    .pipe(share());
  ngOnInit(): void {}
  constructor() {}
}
```

وبعدما اضفنا الدالة share() لنرجع إلى المتصفح ولنشاهد النتيجة، كالتالي:



كما تلاحظ عزيزي المتعلم لا يوجد إلى طلب واحد فقط وتم عرض البيانات بدون أي مشاكل.

3-4-2- shareRelpay():

هذه الدالة بالإضافة إلى اشتمالها على جميع وظائف الدالة share تقوم ايضاً بتخزين البيانات او ما يسمى قيم الObservable لو تم عمل Subscription بعدما تم عمل emit للقيم قبل حدوث هذا الSubscription.

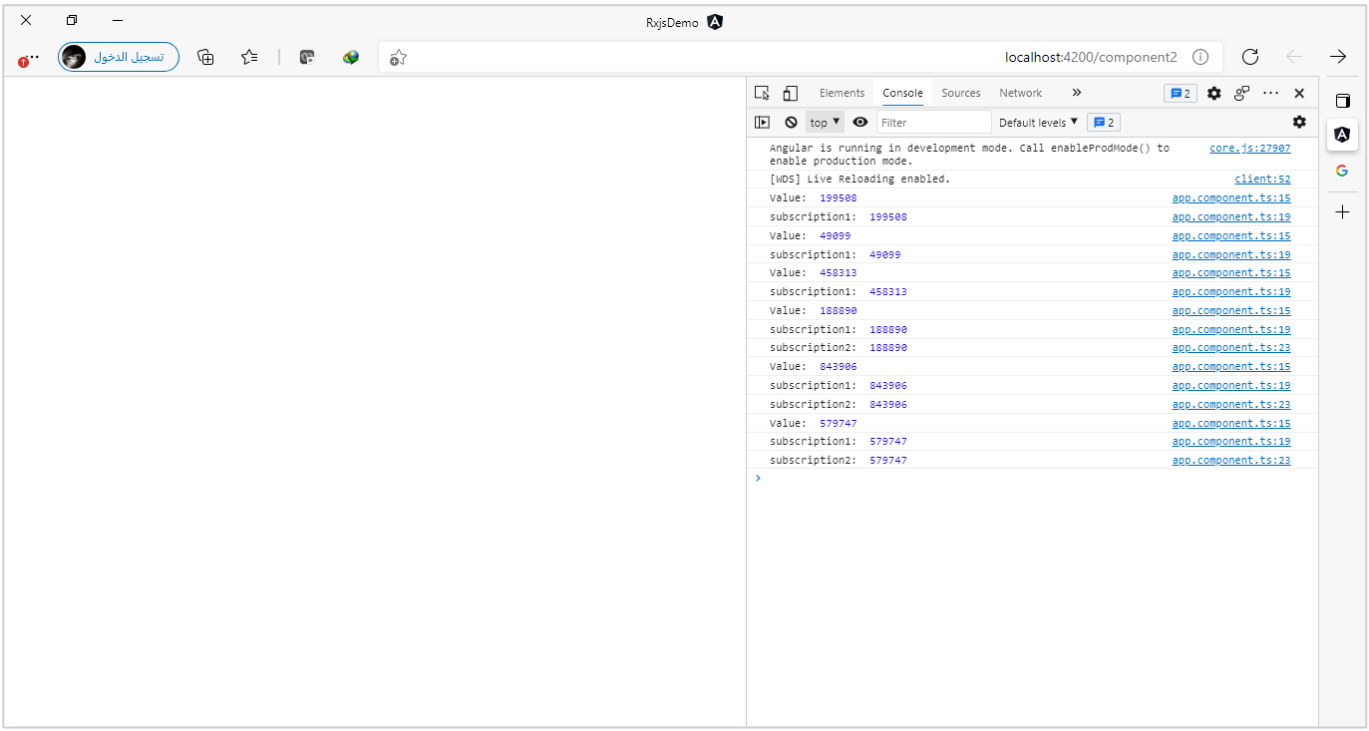
بعبارة أخرى لو كان لدينا Observable وهذا الObservable تم عمل له Multicasting ومن ثم تم عمل subscribe لهذا الObservable وتم عندها ارسال البيانات والقيم من الObserver إلى Subscriber، وبعد فترة معينة تم عمل Subscription آخر جديد، عندها هذا الSubscriber الجديد يكون فقد أي بيانات تم ارسالها قبل لا يعمل Subscription، ومن هنا يأتي دور هذه الدالة حيث تقوم بتخزين البيانات التي تم عمل لها emit وعند عمل Subscription جديد تُرسل البيانات السابقة له، وتعمل ايضاً مشاركة للبيانات الجديدة مع جميع Subscribers.

ولتوضيح لنُعطي المثال التالي والذي سوف نستخدم فيه الدالة share ومن ثم نستخدم الدالة shareReplay لكي نعرف الفرق والاضافة التي اضافتها الدالة الأخيرة.

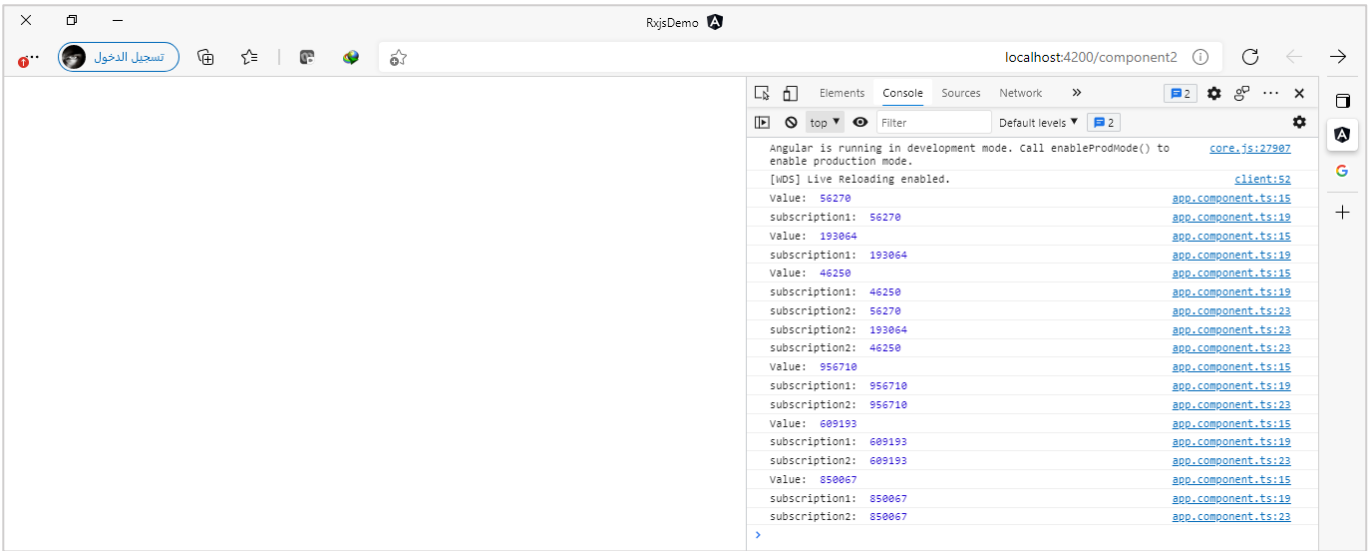
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { map, share, take, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  ngOnInit(): void {
    const interval$ = interval(1000).pipe(
      map(() => Math.floor(Math.random() * 1000000)),
      take(6),
      tap((value) => console.log('Value: ', value)),
      share()
    );
    const subscription1 = interval$.subscribe((x) =>
      console.log('subscription1: ', x)
    );
    setTimeout(() => {
      const subscription2 = interval$.subscribe((x) =>
        console.log('subscription2: ', x)
      );
    }, 3000);
  }
  constructor() {}
}
```

كما نلاحظ لدينا Observable يقوم بعمل emit لقيمة رقمية عشوائية كل ثانية ومن ثم استخدمنا الدالة share لكي نعمل مشاركة لهذه القيم مع جميع Subscribers، وأول subscriber اسميته subscription1 وقام بعمل subscribe مباشرة، اما subscriber الثاني فقد عمل subscribe بعد ثلاث ثواني وهذا معناه ان هنالك ثلاث قيم تم عمل emit لها قبل لا يحدث Subscription الثاني، والآن لنشاهد النتيجة:



كما تلاحظ عزيزي المتعلم عندما قام Subscriber الثاني بعمل subscribe لهذا الObservable جميع القيم التي تم عمل emit لها قبله قد فقدها ولم يعمل عنها أي شيء، لذلك أتت وظيفة الدالة shareReplay والتي تقوم بتخزين القيم ومن ثم عند حدوث أي Subscription جديد وهنالك قيم ستضمن ارسالها له، لذلك لنقم بتغيير الدالة share بالدالة shareReplay، ومن ثم لنشاهد النتيجة في المتصفح، كالتالي:



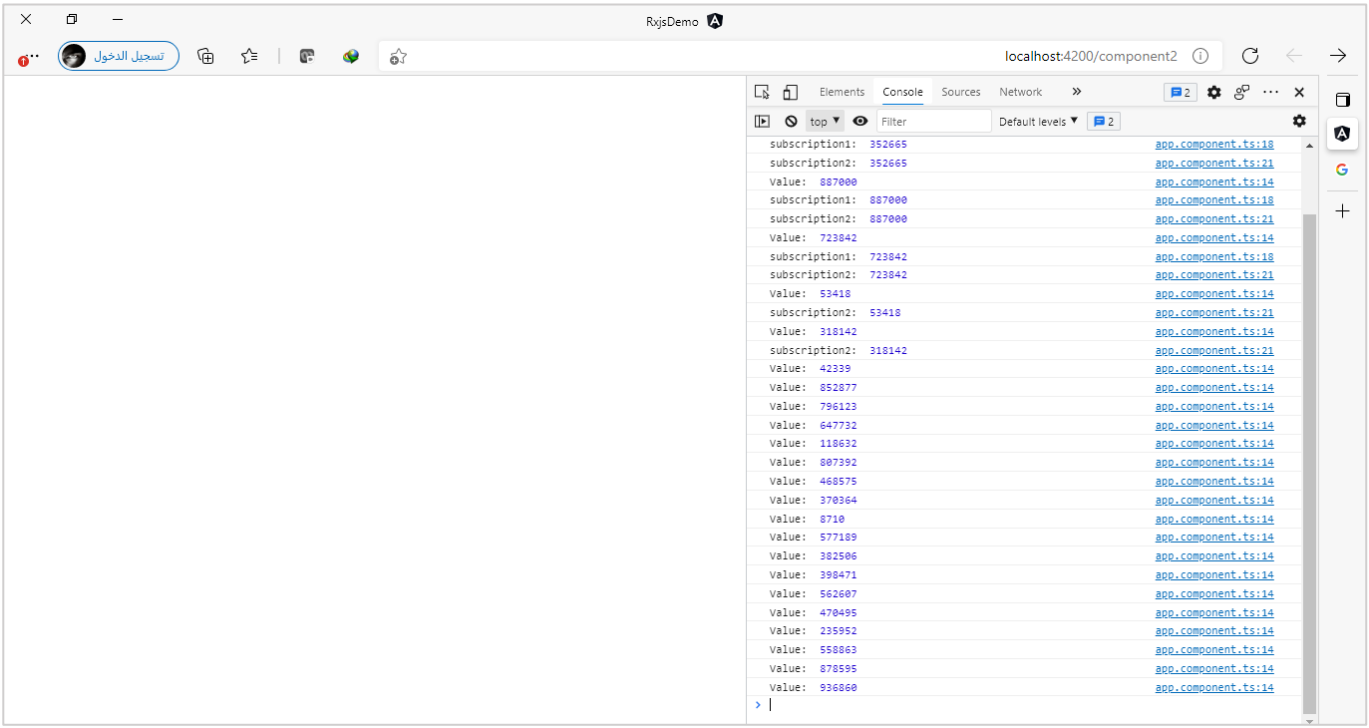
كما نلاحظ في اللحظة التي تم عمل فيها الـSubscription الثاني قامت الدالة shareReplay بإرسال القيم السابقة له، ومن ثم قامت بمشاركة القيم الجديدة مع جميع subscribers.

وتجد الاشارة ان هنالك نقطة مهمة وهي ان هذه الدالة لا تقوم بإيقاف الـObservable في حال عدم وجود أي Subscribers، بمعنى لو كان لدينا Observable يقوم بعمل emit لقيم معينة ويوجد لدينا واحد او اثنين Subscription لهذا الـObservable، ومن ثم بعد فترة زمنية معينة تم عمل Unsubscribe لجميع subscribers إذا كانوا اكثر من واحد او تم عمل unsubscribe للـSubscriber الواحد في حال عدم وجود غيره، ففي هذه الحالة قد يُفضل ان يتم إيقاف هذا الـObservable لأنه لا يوجد أي Subscribers، ولكن ما يحدث هو العكس حيث لا تقوم بإغلاق الـObservable ويستمر بعمل emit لقيمه، وليس معناه ان هذه الطريقة سيئة وانما قد تكون لها استخداماتها، لذلك مكتبة Rxjs اتاحة إضافة خيار نمرره لهذه الدالة وبناءً عليه نستطيع أن نُحدد هل يتم إيقاف الـObservable بمجرد انتهاء جميع الـSubscribers ام لا، وهذا الخيار اسمه refCount وقيمته الافتراضية false وفي حال أردنا ان نجعله true عندها سيتم إيقاف الـObservable بعدما ينتهي جميع Subscriptions، ولتوضيح لنستعرض الكود قبل وبعد استخدام هذا الخيار، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { map, shareReplay, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const interval$ = interval(1000).pipe(
      map(() => Math.floor(Math.random() * 1000000)),
      tap((value) => console.log('Value: ', value)),
      shareReplay()
    );
    const subscription1 = interval$.subscribe((x) =>
      console.log('subscription1: ', x)
    );
    const subscription2 = interval$.subscribe((x) =>
      console.log('subscription2: ', x)
    );
    setTimeout(() => {
      subscription1.unsubscribe();
    }, 3000);
    setTimeout(() => {
      subscription2.unsubscribe();
    }, 5000);
  }
  constructor() {}
}
```

كما تلاحظ عزيزي المتعلم يوجد لدينا اثنين Subscribers لهذا الObservable وبعد ثلاث ثواني تم عمل Unsubscribe للSubscriber الأول، ومن ثم بعد خمس ثواني تم عمل Unsubscribe للObservable الثاني، والآن لنشاهد النتيجة:



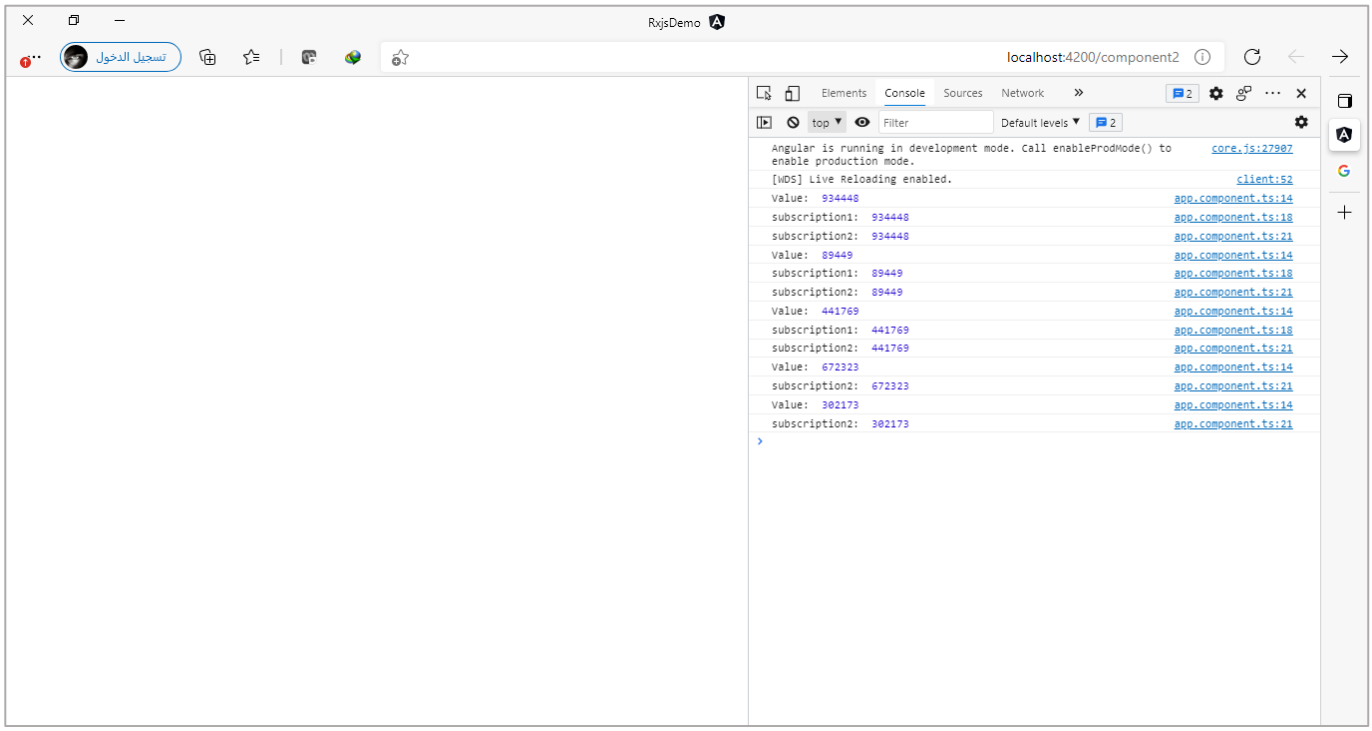
كما نلاحظ على الرغم من انتهاء كلا الSubscriptions إلا ان الObservable لا يزال يعمل emit للقيم الخاصة به، لذلك لتغيير هذه الحالة نستطيع إضافة الخيار refCount إلى الدالة shareReplay على شكل كائن Object ونُعطيها القيمة true، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { map, shareReplay, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  ngOnInit(): void {
    const interval$ = interval(1000).pipe(
      map(() => Math.floor(Math.random() * 1000000)),
      tap((value) => console.log('Value: ', value)),
      shareReplay({ refCount: true })
    );
    const subscription1 = interval$.subscribe((x) =>
      console.log('subscription1: ', x)
    );
    const subscription2 = interval$.subscribe((x) =>
      console.log('subscription2: ', x)
```

```
    );

    setTimeout(() => {
      subscription1.unsubscribe();
    }, 3000);
    setTimeout(() => {
      subscription2.unsubscribe();
    }, 5000);
  }
  constructor() {}
}
```

والنتيجة:



5-3- Utility Operators:

وتُسمى ايضاً Side Effects وهذه الدوال لا تؤثر على الStream او بعبارة أخرى قيم الObservable وانما يتم استخدامها للقيام بمهام معينة بالتوازي مع هذا الObservable، كالدالة tap التي استخدمناها بكثرة في ثنايا هذا الكتاب.

ولها استخدامات متعددة مثل ان نقوم بإظهار صفحة التحميل للمستخدم اثناء بداية الObservable وايضاً إخفاء هذه الصفحة اثناء الانتهاء وجلب البيانات او حدث خطأ ما، او ان نقرأ قيمة معينة من الObservable لتنفيذ مهمة موازية، او تأخير الObservable لفترة زمنية معينة دون التأثير فيه، ...الخ.

وسوف نستعرض اهم هذه الدوال كالتالي:

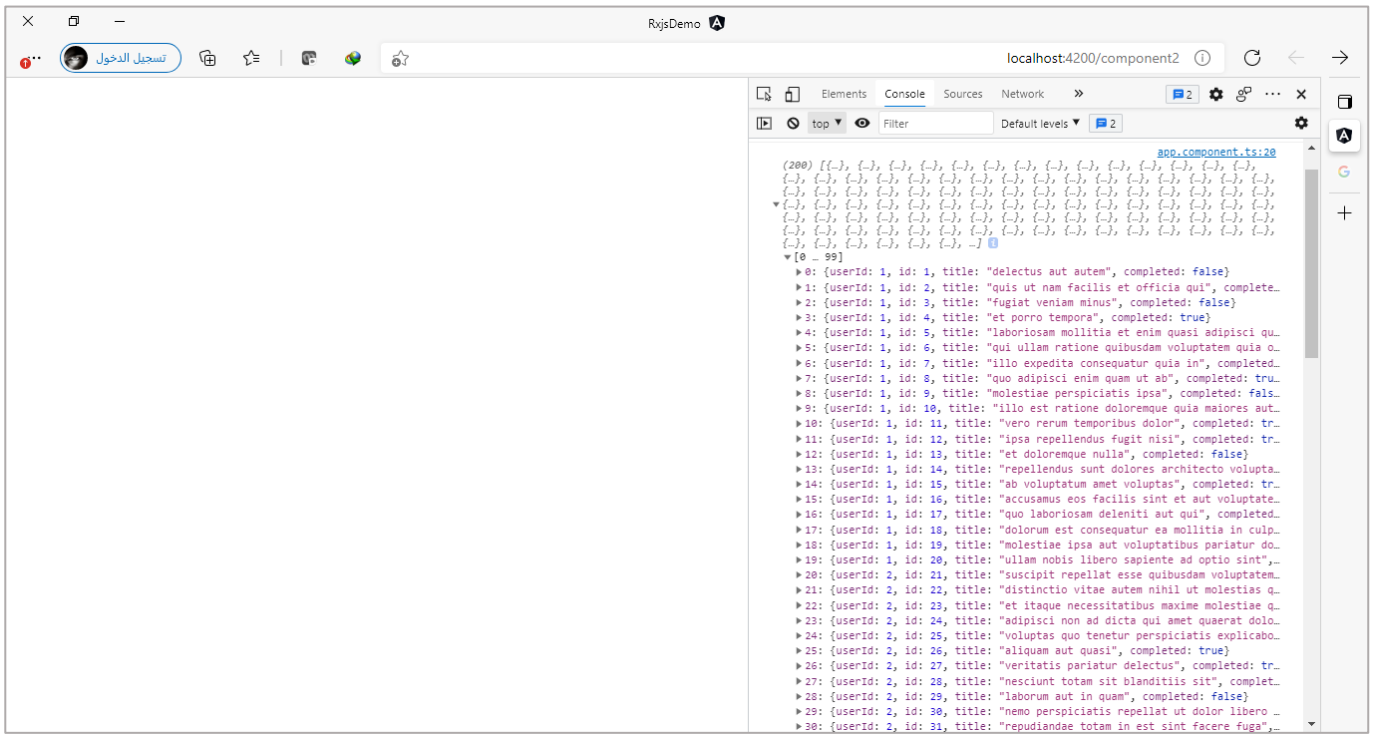
3-5-1- delay() :

اول دالة سوف نستعرضها في هذه المجموعة هي دالة تأخير الObservable لفترة معينة، مع العلم انها لن تؤثر بالStream وانما فقط تقوم بتأخير وصوله إلى Subscribers، وتأخذ باراميتر واحد وهو الوقت بالملي ثانية، كما في المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { delay } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(delay(5000))
      .subscribe(console.log(value));
  }
  constructor() {}
}
```

قمنا بجلب بيانات معينة باستخدام دالة ajax ورابط API معين ومن ثم مررنا هذا الObservable على دالة delay حيث أخرنا وصول هذا الStream للSubscriber لمدة خمس ثواني.



3-5-2- tap():

لعل هذه الدالة هي أشهر دالة من دوال هذه المجموعة وأكثرها شيوعاً واستخداماً وكانت تسمى سابقاً do وتم تعديل اسمها في الإصدار السادس من مكتبة rxjs وما يليه من إصدارات بالاسم tap.

ولقد استخدمناها كثيراً وهنالك أمثلة كثيرة سابقة استخدمنا فيها هذه الدالة، وسوف نعطي هنا ايضاً مثال آخر وهو التعديل على المثال الذي قمنا باستعراضه سابقاً في دالة delay بحيث نُظهر رمز التحميل والانتظار للمستخدم خلال انتظاره لعرض البيانات، كما في المثال التالي:

ملف app.component.html

```
<div *ngIf="loading === true">
  <img src="../assets/loader.jpg" />
</div>
```

تجدر الإشارة انني قمت بجلب صورة من الانترنت ووضعتها في مجلد assets في مشروعنا وهذه الصورة هي عبارة عن loader الذي نُريد اظهاره للمستخدم بحيث لا يظهر إلى إذا كان المتغير loading يساوي true، والآن لنذهب إلى ملف class لهذا الcomponent ونعرف المتغير السابق ونضيف باقي Logic البرمجي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { delay, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  loading = false;
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(
        tap(() => (this.loading = true)),
        delay(5000)
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => (this.loading = false),
        complete: () => (this.loading = false),
      });
  }
  constructor() {}
}
```

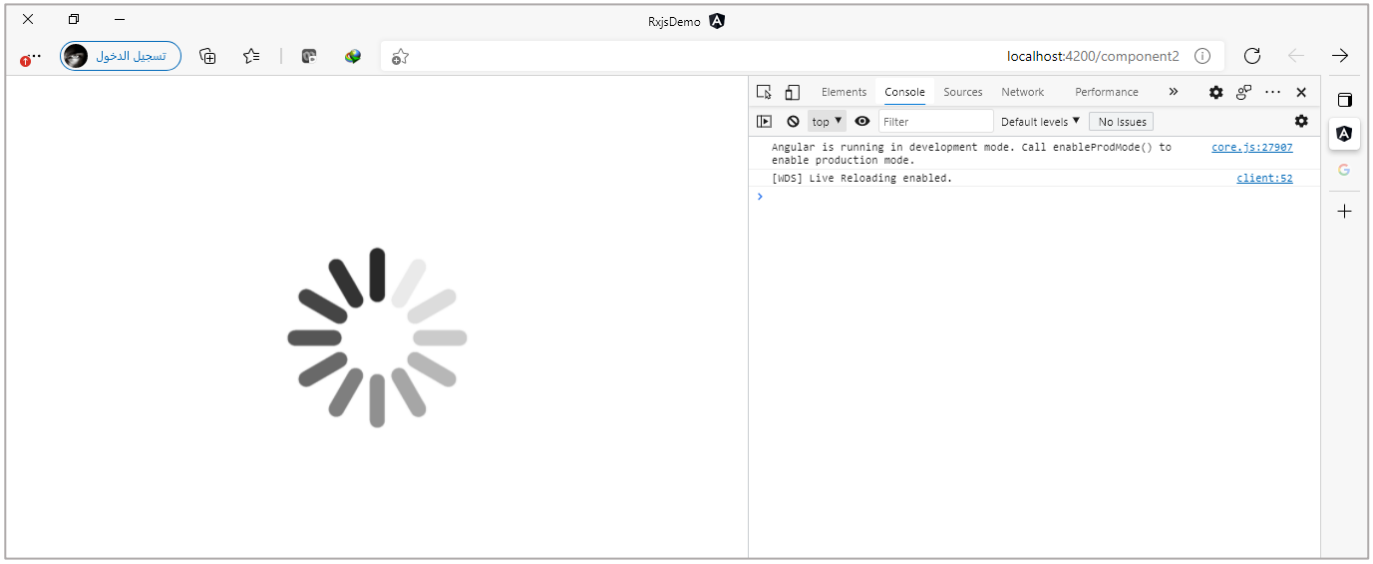
كما تلاحظ عزيزي المتعلم في Stream قمنا بتغيير قيمة المتغير إلى true وهذا معناه اظهار صورة التحميل للمستخدم، ومن ثم في Subscription قمنا بإرجاعه إلى false سواء إذا اكتمل هذا الObservable او حدث خطأ معين.

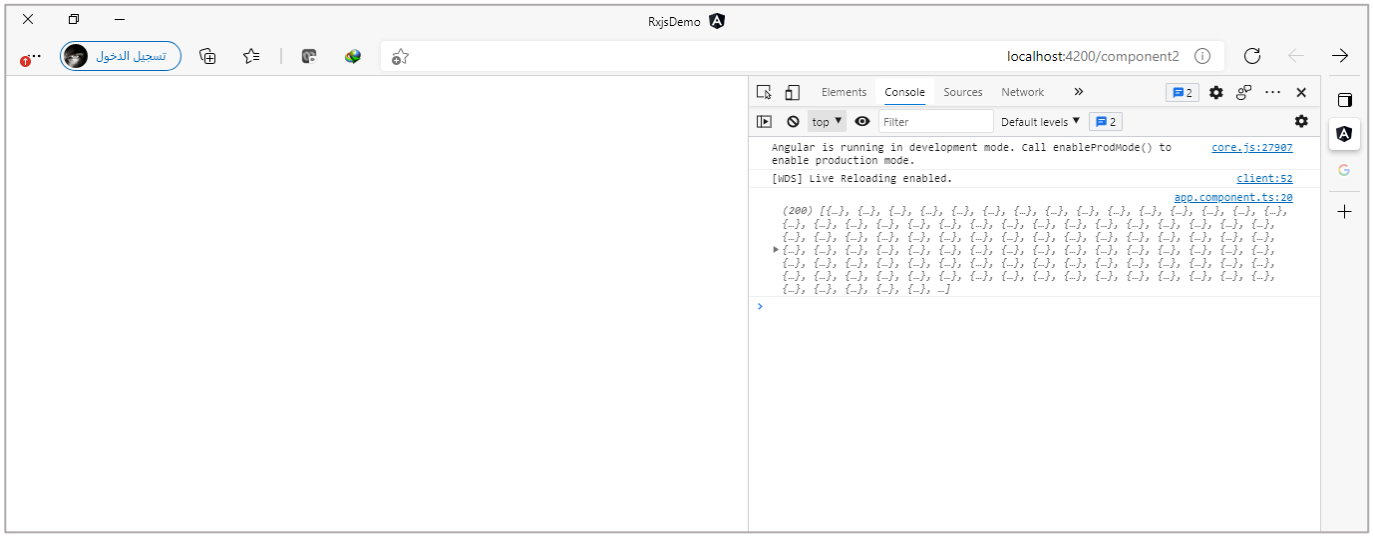
واخيراً نضيف بعض التنسيقات لهذه الصورة في ملف style، كالتالي:

ملف app.component.scss

```scss
div {
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 80vh;
}

img {
  width: 200px;
}
```

اما الآن لنشاهد النتيجة في المتصفح:





3-5-3- finalize():

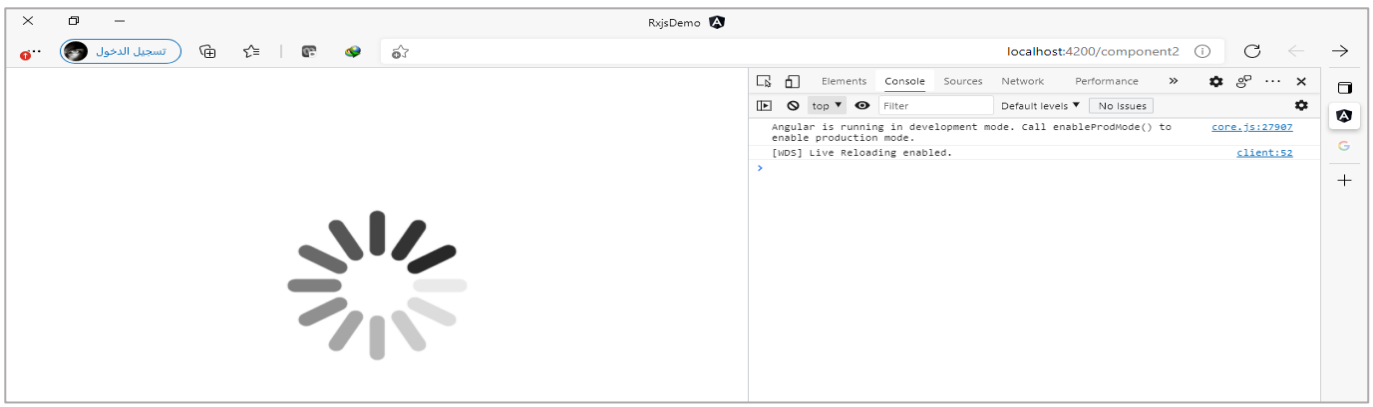
اما هذه الدالة فيتم استدعائها في حالتين الأولى عند حدوث خطأ في الStream والحالة الثانية عند اكتماله.

ولتوضيح لنقوم بالتعديل على المثال السابق في الدالة tap بحيث بدلاً من ان نُغير قيمة المتغير loading إلى false في موقعين في Subscription وهما في حالة وجود خطأ والحالة الثانية في حالة اكتمال هذا الObservable، نقوم بتغييره فقط في الدالة finalize فقط لأنها كما قلنا سابقاً يتم استدعائها في حالة الخطأ وفي حالة الاكتمال.

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { delay, finalize, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  loading = false;
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(
        tap(() => (this.loading = true)),
        delay(5000),
        finalize(() => (this.loading = false))
      )
      .subscribe(console.log);
  }
  constructor() {}
}
```

والآن لنشاهد النتيجة في المتصفح كالتالي:



سوف تلاحظ نفس النتيجة في مثال tap ولكن هنا تم كتابة logic بطريقة اكثر احترافية وعمومية.

6-3- Error Handling Operators:

من المهارات المهمة التي يجب على كُل مبرمج ومطور ان يتقنها هي التعامل مع الإخطاء بأي لغة برمجة ومكتباتها واطر العمل الخاصة بها، ومكتبة Rxjs ليست بمعزل عن هذا الجانب لذلك قدمت لنا مجموعة من الطرق والأساليب والدوال التي تُساعدنا وتُسهل علينا التعامل مع الإخطاء مهما كانت ومحاولة حلها.

وقد تطرقنا سابقاً في ثنايا هذا الكتاب إلى مجموعة من الصور لتعامل مع هذه الأخطاء كالجزء الخاص بالتعامل مع الأخطاء بشكلها البسيط، او من خلال بعض الدوال كالدالتين throwError او onErrorResumeNext.

اما هنا سنتطرق لطرق ودوال أخرى لتعامل مع الأخطاء بشكل أكثر احترافية وهذا لا يعني ان ما قيل سابقاً فيه أخطاء او انه لا يصلح ولكن دائماً في عالم البرمجة ضع في ذهنك انه مهما تعددت الطرق فالهدف واحد ومن هذا المنطلق اسلك دائماً الطريق الأقصر والأقل مجهود مع الحفاظ على الأداء الجيد وسرعة التطبيق، ونحن هنا في هذا الكتاب نستعرض أغلب الطرق واترك لك عزيزي المتعلم ان تختار الطريق المُفضل لديك.

ونستطيع ان نقول وبشكل عام هنالك طريقتين رئيسيتين لتعامل مع الأخطاء في مكتبة Rxjs وهما:

- عن طريق الSubscription وهو ما تطرقنا إليه سابقاً في الفصل الأول من هذا الكتاب تحت عنوان معالجة الأخطاء بشكله البسيط عن طريق Angular وتقنيات الObservable.
- عن طريق نفس Stream وقبل وصول القيم إلى الSubscription وبالتحديد في دالة Pipe وهو ما سنتطرق له بإذن الله في هذه الجُزئية.

مع ان بعض المراجع تُفضل الطريقة الثانية بسبب أسلوب Clean Code ولسهولة الفهم والتطوير وإعادة الاستخدام على مستوى التطبيق كامل، ولكن رأيي الشخصي ليس هنالك طريقة افضل من طريقة وانما نستخدم الطريقة التي تلبي احتياجنا ولو كان الاحتياج الدمج بين هاتين الطريقتين.

وباستخدامنا للطريقة الثانية والتي هي معالجة الإخطاء داخل الدالة Pipe هنالك مجموعة من الآليات والأساليب التي تُقدمها هذه الطريقة للتعامل مع الأخطاء، يمكن اجمالها في النقاط التالية:

- إعادة المحاولة لمعالجة الخطأ.
- التقاط الخطأ ومعالجته (بعمل رمي throw لهذا الخطأ – تحويل الخطأ على شكل Observable).

ويتم تحقق هذه النقطتين من خلال مجموعة من الدوال الجاهزة التي تُقدمها لنا مكتبة Rxjs وهي دوال retry وretryWhen وcatchError، والتي سوف نتطرق لها لاحقاً بإذن الله.

وقبل الاسترسال في كيفية معالجة الأخطاء يجب ان ننتبه إلى نقطتين مهمين، النقطة الأولى انه عند حدوث أي خطأ فإن دالة complete في Subscription لن يتم استدعائها وانما سيتم استدعاء دالة error فقط، والنقطة الثانية التي يجب ان نُشير إليها هي ماذا يحدث للObservable عند حدوث خطأ ما؟ والذي يحدث ان هذا الObservable وبشكل افتراضي

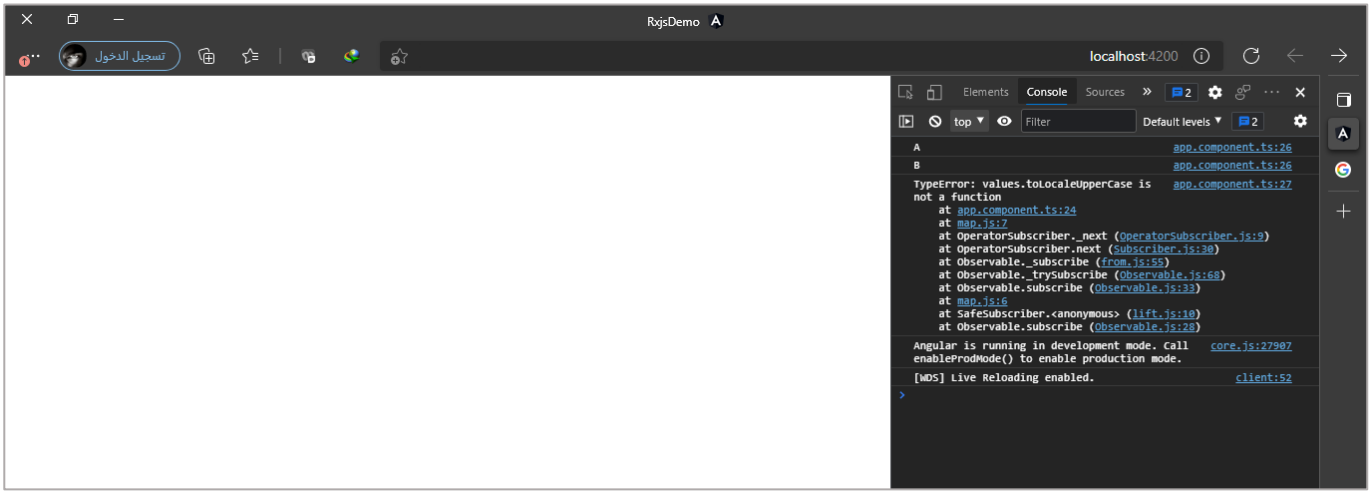
عند حدوث أي خطأ سيقوم مباشرة بإنهاء نفسه وإيقاف أي قيم تأتي بعد هذا الخطأ. لذلك في حال اردنا ان نُغير هذا السلوك الافتراضي فعندها سنحتاج إلى دوال معالجة الأخطاء سابقة الذكر والتي سوف نستعرضها بشيء من التفصيل.

ولتوضيح ما قيل سابقاً تستطيع عزيزي المتعلم الاطلاع على المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    of('a', 'b', 1, 'c')
      .pipe(map((values: string) => values.toLocaleUpperCase()))
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!'),
      });
  }
  constructor() {}
}
```

كما هو ملاحظ اضفنا خطأ متعمد في القيم وهو الرقم 1 حيث عنده سيحدث خطأ وهذا الخطأ مصدره الدالة toLocalUpperCase لأنها تتعامل مع القيم النصية فقط لذلك سيحدث الخطأ، وسنلاحظ عندها ان النتيجة ستكون قراءة الحرف a والحرف b ومن ثم سيحدث الخطأ ويتم استدعاء الدالة error وإيقاف هذا الObservable وعدم قراءة أي قيم أخرى، كالتالي:



اما الآن لنستعرض هذه الدوال ولنبدأ بأول دالة وهي دالة retry.

3-6-1- retry():

وهذه اول دالة من دوال معالجة الأخطاء من خلال مكتبة Rxjs، ونستخدمها عادة مع الأخطاء التي نتوقع انه مع إعادة الاتصال مع هذا الObservable لمرة او عدد من المرات سيتم تخطي هذا الخطأ وجلب البيانات بشكل صحيح، مثل أخطاء شبكة الانترنت قد يحدث ضعف او انقطاع لحظي لثواني في شبكة الانترنت اثناء جلب البيانات او قد يكون هنالك ضغط على السيرفر مما يؤثر على جلب البيانات، لذلك نستخدم هذه الدالة لمحاولة جلب هذه البيانات مرة أخرى دون الحاجة لإظهار رسالة خطأ للمستخدم ونطلب منه تحديث الصفحة او الضغط على الزر لأعاده المحاولة.

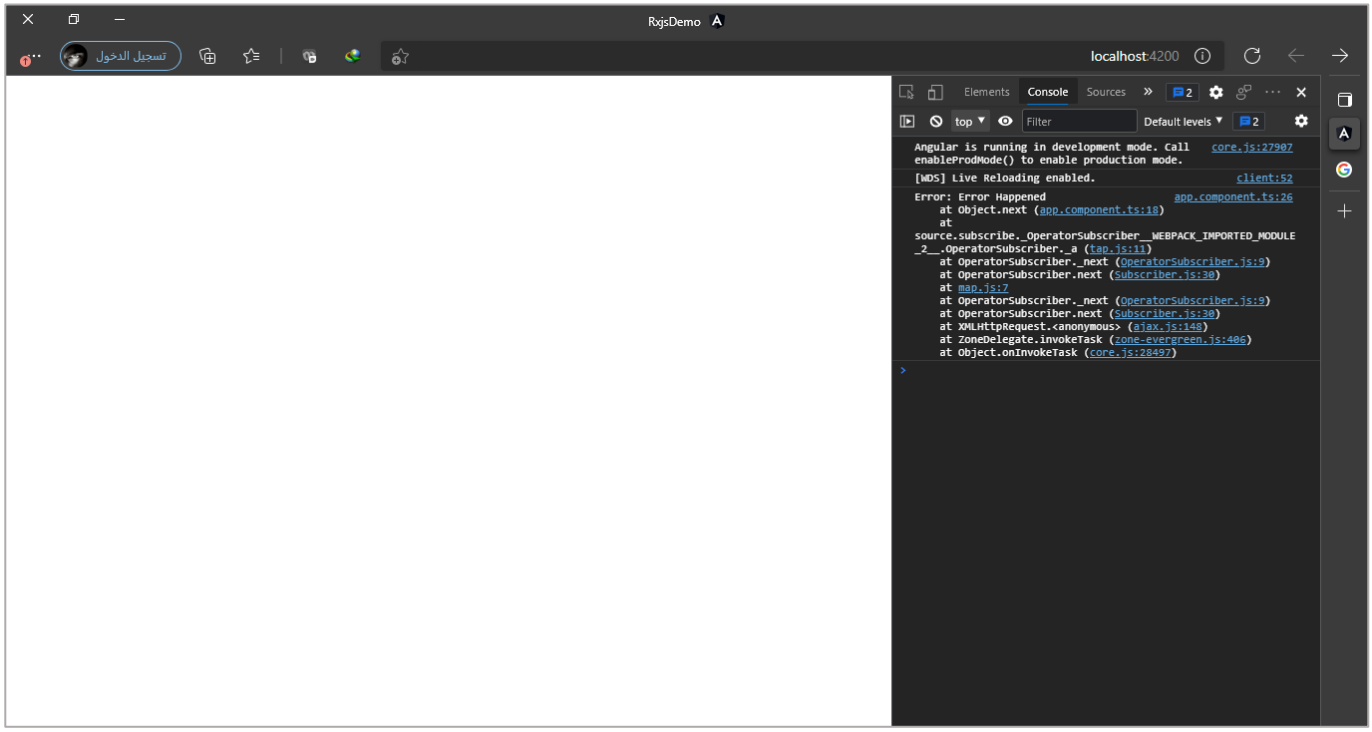
وتستقبل هذه الدالة باراميتر اجباري واحد وهو قيمة رقمية تمثل عدد المحاولات، وطريقة عملها بسيطة حيث عند وقوع أي خطأ تقوم بعمل Unsubscribe للObservable وتُعيد عمل Subscribe من جديد لهذه الObservable، مع العلم ان هذا Subscription يتم بشكل آلي أي كل الذي نعمله ان نستدعي هذه الدالة ونمرر لها قيمة رقمية تمثل عدد المحاولات.

ولتوضيح لنستعرض المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  private count = 0;
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(
        tap(() => {
          this.count++;
          if (this.count === 1) {
            throw Error('Error Happened');
          }
        })
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
  constructor() {}
}
```

كما نلاحظ الأسطر البرمجية السابقة قمنا عن طريقها باستدعاء بيانات معينة عن طريق API ومن ثم قمنا بمحاكاة ان اول مرة سيتم عمل throw لخطأ معين لذلك لنشاهد النتيجة:



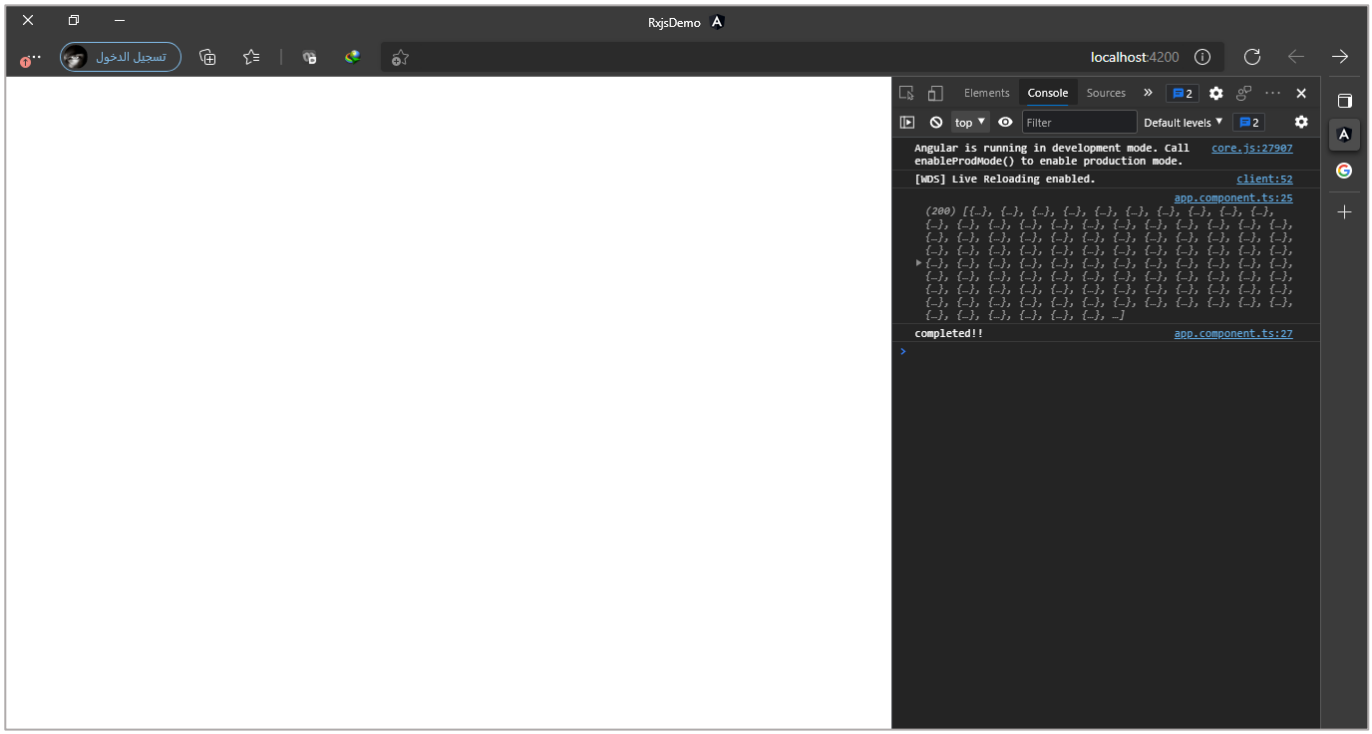
كما تلاحظ عزيزي المتعلم ظهر لنا خطأ غير مقصود لذلك نستطيع ان نُعيد المحاولة لجلب البيانات باستخدام الدالة retry، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { retry, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  private count = 0;
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(
        tap(() => {
          this.count++;
          if (this.count === 1) {
            throw Error('Error Happened');
          }
        }),
        retry(2)
      )
```

```
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
  constructor() {}
}
```

انا هنا قمت بتمرير العدد 2 لدالة retry وكأنني أقول قم بالمحاولة مرة وفي حال الفشل ايضاً حاول مرة أخرى، مع العلم انه يمكنك إضافة عدد المحاولات التي تُريد بحسب احتياجك، وفي حال اردت ان تكون المحاولات لا نهائية نمرر الكلمة infinity كباراميتر لهذه الدالة.



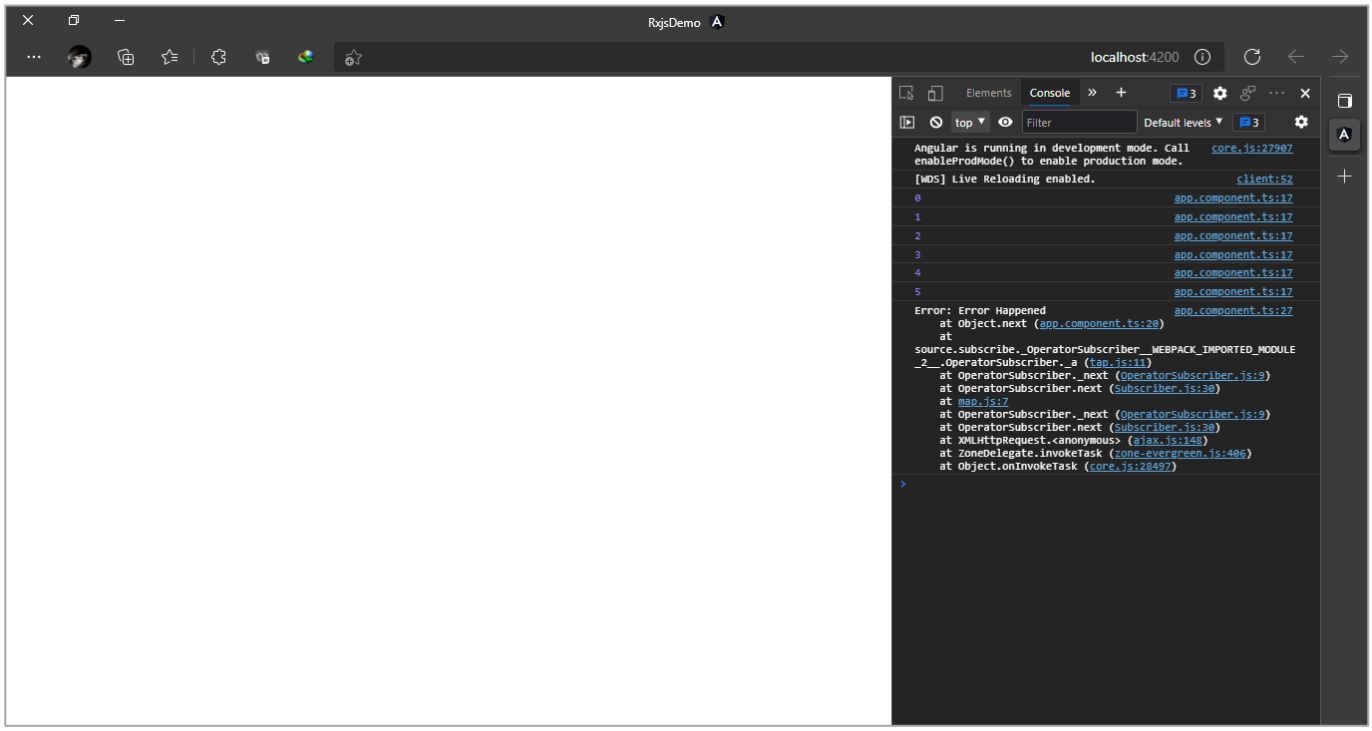
اما في حالة استنفاذ كافة المحاولات ولايزال الخطأ يحدث فعندئذً سيتم رمي خطأ Throw Exception، ولتوضيح لنقوم بتعديل المثال السابق بحيث يستمر الخطأ رغم عدد المحاولات، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ajax } from 'rxjs/ajax';
import { retry, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  private count = 0;
  ngOnInit(): void {
```

```
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(
        tap(() => {
          console.log(this.count);
          this.count++;
          if (this.count >= 1) {
            throw Error('Error Happened');
          }
        }),
        retry(5)
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
  constructor() {}
}
```

سوف نلاحظ على الرغم من اننا جعلنا عدد المحولات 5 إلى انها جميعاً سوف تفشل بجلب البيانات ومعالجة الخطأ، فلذلك عندها سيتم رمي خطأ Throw Exception، وايضاً اضفت سطر جديد لعرض كم مرة تمت الإعادة وذلك من خلال الاستفادة من المتغير count والفائدة منه للتوضيح فقط لا غير، والآن لنشاهد النتيجة، كالتالي:



كما تلاحظ عزيزي المتعلم تمت إعادة المحاولة لخمس مرات بالإضافة إلى المحاولة الأولى وبذلك ستصبح عدد المحاولات ست محاولات، ومن ثم تم رمي الخطأ.

3-6-2- retryWhen():

اما هذه الدالة فوظيفتها مشابهه لدالة السابقة وتختلف عنها بأنها تستقبل دالة Function على شكل باراميتر وتُعيد Observable، وفي العادة يتم استخدام هذه الدالة لعدة حالات، يمكن اجمالها من خلال النقاط التالية:

- في حال أردنا أن ننتظر فترة زمنية معينة قبل أعادة المحاولة بدلاً من تمرير عدد المحاولات كما هو الحال في الدالة السابقة retry، ومن ثم يتم رمي الخطأ في حال الفشل وتفعيل دالة completeفي Subscriber في حالة اكتمال Observable.
- في حال أردنا ان نُعيد المحاولة بعد معالجة الخطأ لكي نستكمل قراءة البيانات بدون إيقاف Observable، وبذلك يستمر Observable إلى ان يكتمل ومن ثم يتم تفعيل دالة complete في Subscriber.
- في حال أردنا ان نكتب اسطر إضافية مع تحديد عدد المحاولات.

وسوف نتطرق على جميع هذه الحالات من خلال الأمثلة، كالتالي:

3-6-2-1- الحالة الأولى:

في هذه الحالة سوف يتمر تمرير وقت معين قبل أعادة المحاولة مرة أخرى بدلاً من تمرير عدد المحاولات، وكأننا نقول في حالة وقوع خطأ قم بعمل إعادة محاولة بعد فترة زمنية معينة وفي حال الفشل قم برمي خطأ وتفعيل دالة errorفي Subscriber واما في حالة النجاح استكمل Observable إلى ان يكتمل ومن ثم فعل الدالة complete.

مع العلم ان هذه الدالة تستقبل دالة Function وهذه الدالة Function هي الأخرى تستقبل باراميتر وهذا الباراميتر عبارة عن Observable ايضاً وهو عبارة عن الخطأ الذي وقع ويتم استقباله هنا، وايضاً هي تُعيد Observable أيً كان هذا Observable، ولكن نحن في مثالنا التوضيحي سنعيد وقت على شكل Observable وذلك من خلال الاستفادة من الدالة delay، كالتالي:

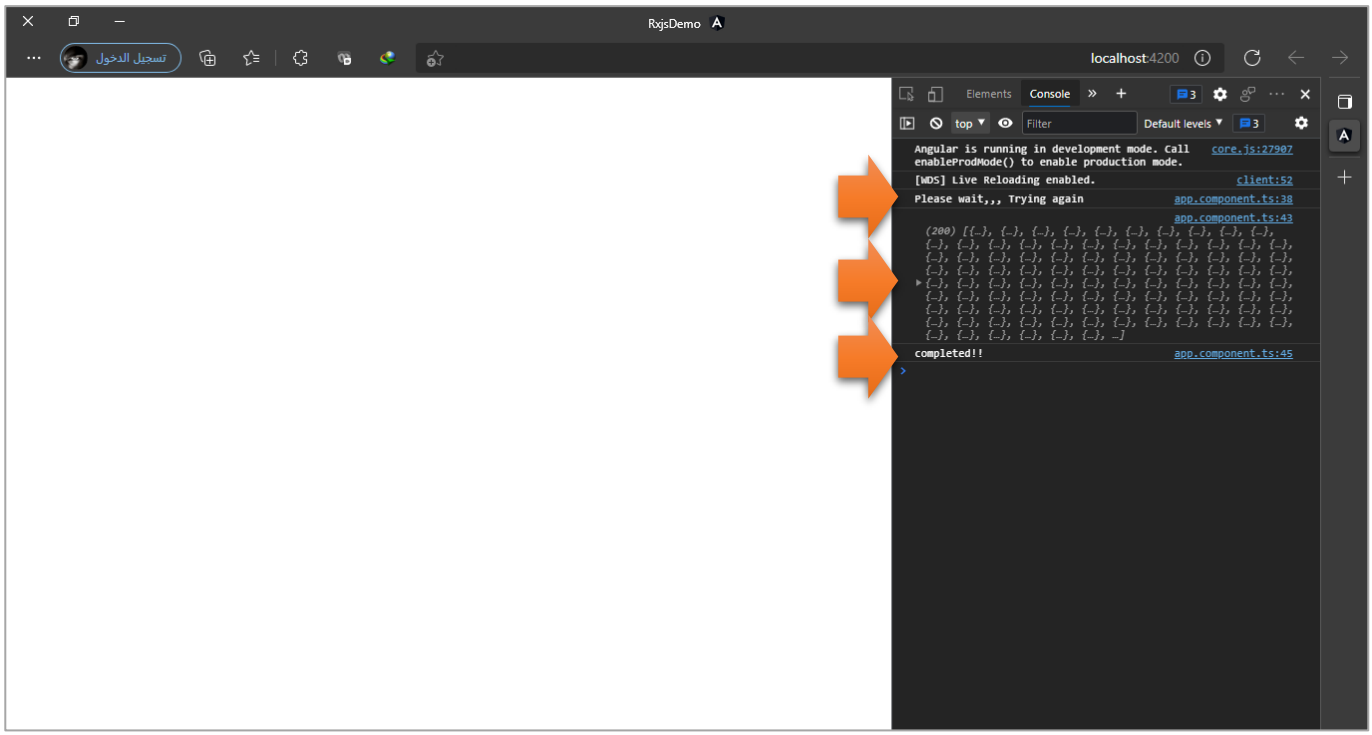
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { delay, retryWhen, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  private count = 0;
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todos')
      .pipe(
```

```
        tap(() => {
          this.count++;
          if (this.count === 1) {
            throw Error('Error Happened');
          }
        }),
        retryWhen((errors) => {
          console.log('Please wait,,, Trying again');
          return errors.pipe(delay(2000));
        })
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

لو تلاحظ عزيزي المتعلم ان هذا المثال هو نفسه المثال السابق في الدالة retry ولكن هُنا استخدمنا الدالة retryWhen ومررنا لها Function وهذه الFunction ايضاً مررنا لها باراميتر حيث يمثل الخطأ الذي حصل.

والذي قمنا به هو اظهار رسالة معينة في الconsole ومن ثم قمنا بإعادة العObsrervable بعد تأخيره لفترة زمنية معينة امتد إلى ثانيتين، والآن لنشاهد النتيجة في المتصفح:



كما نلاحظ عند وقوع الخطأ تم عرض الرسالة في console ومن ثم بعد ثانيتين تم إعادة المحاولة وفي هذه المرة نجحت المحاولة وتم قراءة وعرض البيانات وعند الاكتمال تم استدعاء الدالة complete في Subscriber.

3-6-2-2- الحالة الثانية:

اما في هذه الحالة فنستخدم الدالة retryWhen ليس فقط لتكرار المحاولة ولكن لمعالجة الخطأ ومن ثم استمرار الObservable بدون رمي خطأ او اغلاقه.

وهذه الحالة يحكمنا فيها طبيعة الخطأ فإذا كان من الممكن معالجته في التطبيق وبنفس الوقت حاجتنا تكمُن في استكمال الObservable وعدم إيقافه فعندها نلجأ إلى هذه الحالة،

ولتوضيح لنُعطي المثال التالي والذي هو نفس المثال الذي طرحناه سابقاً في الدالة debounceTime حيث سنقوم بقراءة الObservable والذي هو عبارة عن القيمة النصية التي سيدخلها المستخدم في مربع النص، والخطأ الذي سوف نفتعله في هذا المثال هو في حالة قام المستخدم بكتابة ارقام في مربع النص فعندها سيتم رمي خطأ، لذلك سنستخدم هذه الدالة لمعالجة الخطأ وحذف القيمة الخطأ من مربع النص مع استكمال التواصل مع السيرفر لجلب البيانات وقراءتها، كالتالي:

ملف app.component.html

```
<div class="text-center p-5">
  <input #input [(ngModel)]="str" class="w-100" />
</div>
```

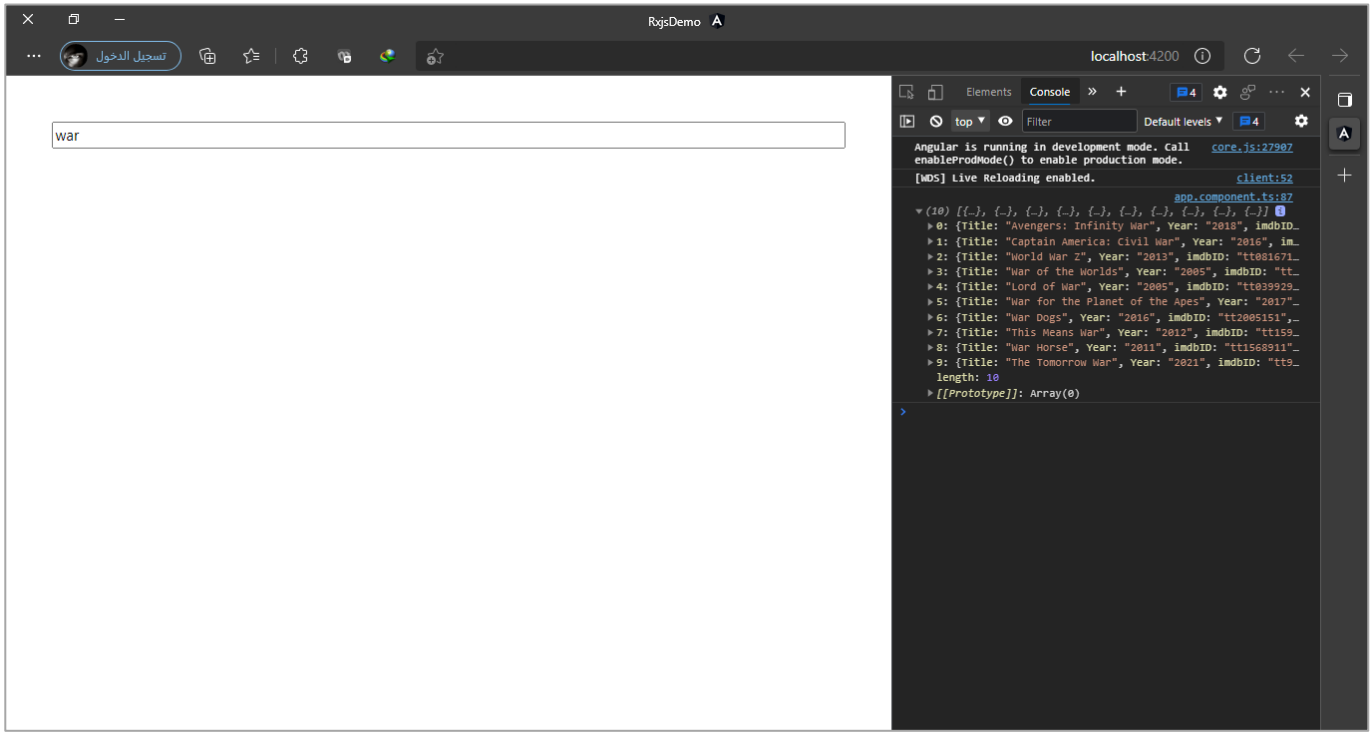
وفي ملف class نضيف Logic البرمجي التالي:

ملف app.component.ts

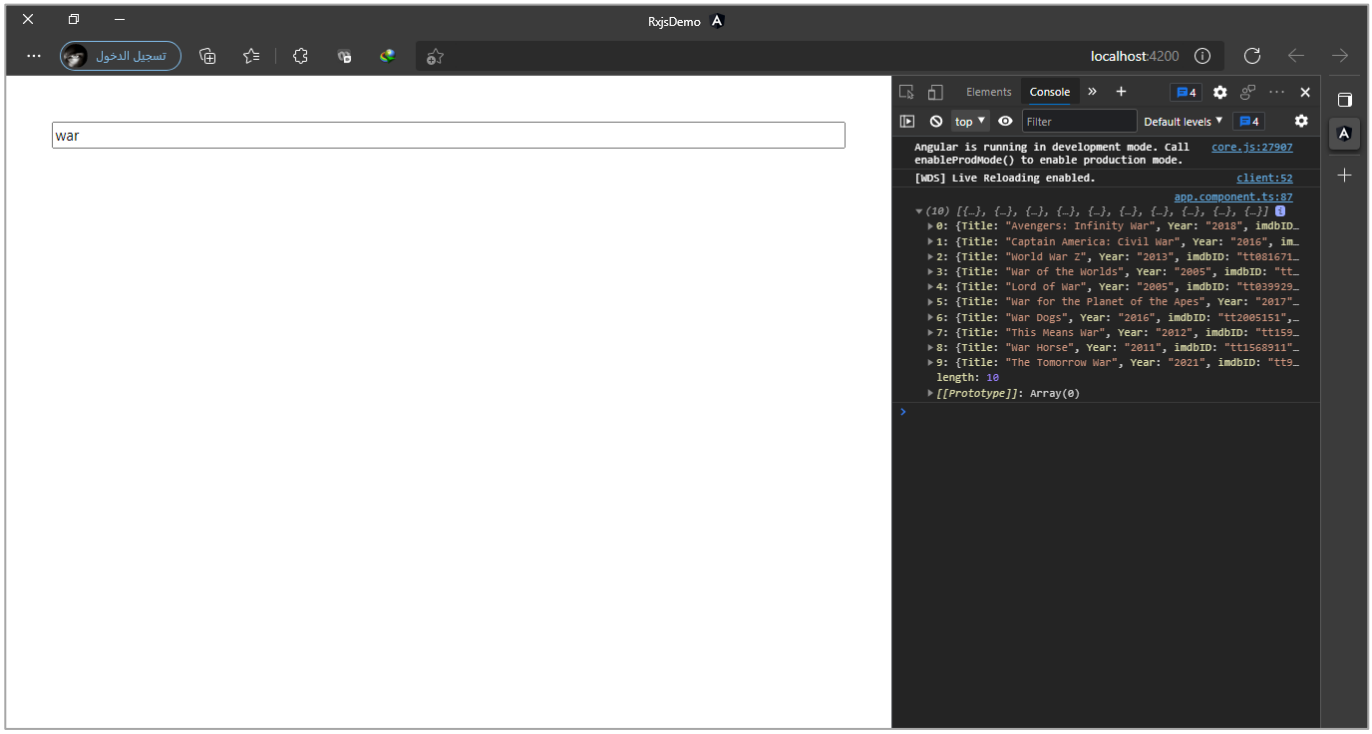
```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { fromEvent, of, throwError } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { catchError, debounceTime, distinctUntilChanged, map, retryWhen, switchMap, tap,
} from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  @ViewChild('input', { static: true }) private input: ElementRef<HTMLInputElement>;
  private char: string;
  str = '';

  ngOnInit(): void {
    fromEvent(this.input.nativeElement, 'keyup')
      .pipe(
        switchMap((key: any) => {
          const { value } = key.target;
          if (key.keyCode >= 48 && key.keyCode <= 57) {
            this.char = String.fromCharCode(key.keyCode);
            return throwError(() => {
```

```
            return value;
          });
        } else {
          return this.str;
        }
      }),
      debounceTime(1000),
      distinctUntilChanged(),
      switchMap(() => {
        return ajax({
          url: `http://www.omdbapi.com/?s=${this.str}&apikey=e8067b53`,
          method: 'get',
          queryParams: this.str,
          crossDomain: true,
        });
      }),
      map((res: any) => res.response),
      retryWhen((obs) => {
        return obs.pipe(
          map((value) => {
            this.str = value.replace(this.char, '');
            return this.str;
          })
        );
      })
    )
    .subscribe({
      next: (response) => console.log(response.Search),
      error: (error) => console.log(error),
      complete: () => console.log('complete!!'),
    });
  }
  constructor() {}
}
```

لا تهتم عزيزي المتعلم بكثرة الاسطر البرمجية فبعضها الهدف منها افتعال خطأ.لنعرف كيفية التعامل معه وبعضها الآخر قمنا بالتطرق له سابقاً في مثال debounceTime والذي يهمنا في الحقيقة ما تم التأشير عليه بالسهم حيث استقبلنا الخطأ على شكل Observable من خلال الباراميتر obs (لك حرية اختيار الاسم الذي تُريده) وعن طريقه قرأنا القيمة ومن ثم عالجنا الخطأ الحاصل بهذه القيمة وهو حذف الرقم ومن ثم قمنا بإعادة القيمة بعد معالجتها لعرضها في مربع النص، وبذلك تكون النتيجة كالتالي:



الآن قم بإدخال قيمة رقمية ولنرى ما الذي سوف يحدث؟



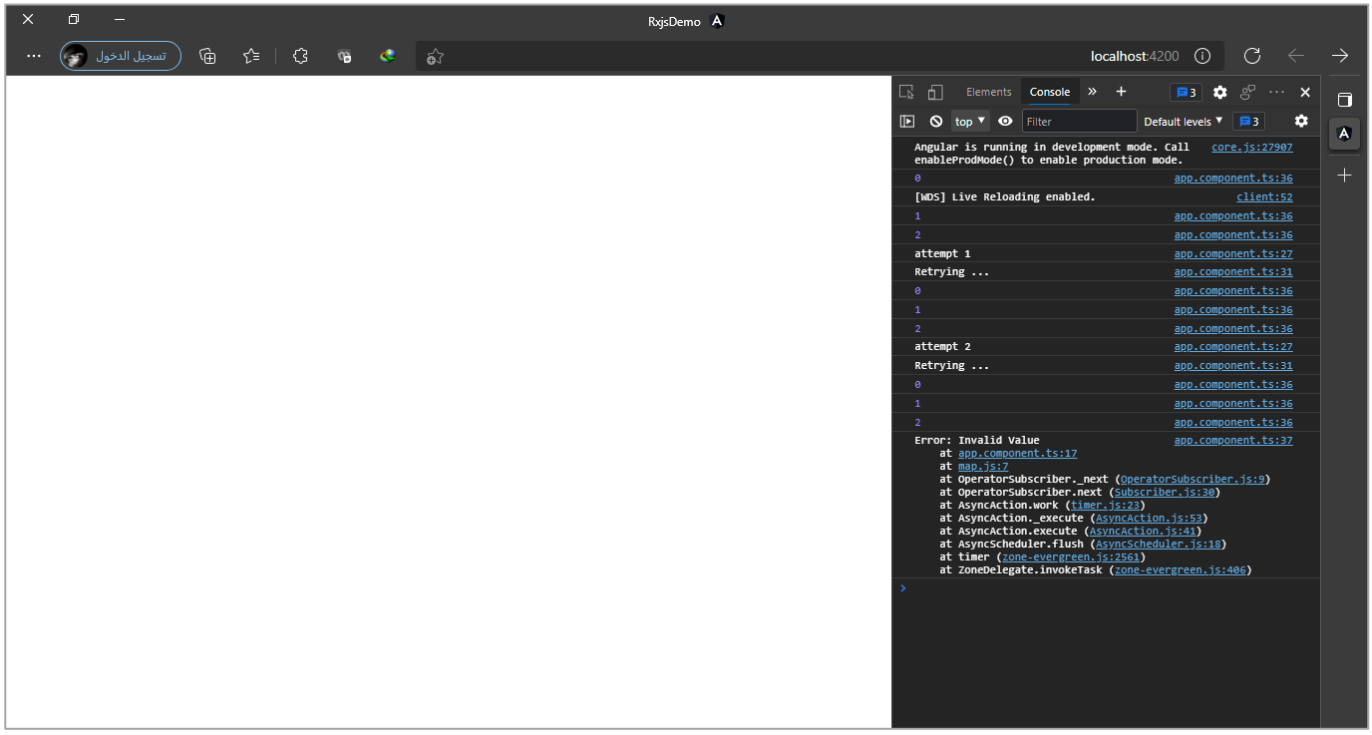
سوف تلاحظ انه سيتم معالجة الخطأ مباشرة وذلك عن طريق مسح القيمة الرقمية دون الحاجة إلى اجراء أي طلب إلى السيرفر.

3-2-6-3- الحالة الثالثة:

اما هذه الحالة فهي إذا أردنا ان نجمع بين وظيفة retry وهي تحديد عدد المحاولات مع إمكانية إضافة اسطر برمجية إضافية، ولتوضيح لنستعرض المثال التالي:

app.component.ts ملف

```
import { Component, OnInit } from '@angular/core';
import { interval } from 'rxjs';
import { delay, map, retryWhen, scan, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    interval(1000)
      .pipe(
        map((val) => {
          if (val > 2) {
            throw new Error('Invalid Value');
          }
          return val;
        }),
        retryWhen((error) =>
          error.pipe(
            scan((acc, err) => {
              if (acc > 2) {
                throw err;
              }
              console.log('attempt ' + acc);
              return acc + 1;
            }, 1),
            delay(3000),
            tap(() => console.log('Retrying ...'))
          )
        )
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```



catchError() -3-6-3:

هذه الدالة تختلف عن الدالتين السابقتين انها لا تُعيد عمل subscribe للObservable الأساسي عند وقوع خطأ وإنما تقوم بمهمتين الأولى تُعيد خطأ او تُعيد Observable جديد يختلف عن الأساسي.

وتستقبل بارامترين الأول اجباري وهو عبارة عن الخطأ الذي وقع، والثاني اختياري وهو عبارة عن الObservable الأساسي، ولكن يجب الحذر عند استخدام البارامتر الثاني واعادته لأنه حينها سوف تدخل في ما يسمى Infinite Loop بحيث يحدث الخطأ ويتم التقاطه بواسطة هذه الدالة ويتم إعادة return الObservable الأساسي وعندها ايضاً يحصل الخطأ مرة أخرى وايضاً يتم التقاطه، وهكذا إلى مالا نهاية.

ومما قيل سابقاً نستطيع القول ان وظيفة هذه الدالة تتلخص في مهمتين، هما:

- التقاط الخطأ وإعادة رمي نفس الخطأ او أي خطأ آخر.
- التقاط الخطأ وإعادة Observable جديد مهما كان هذا الObservable الجديد.

ولتوضيح سوف نُعطي مثال على كلا المهمتين، كالتالي:

ملف app.component.ts

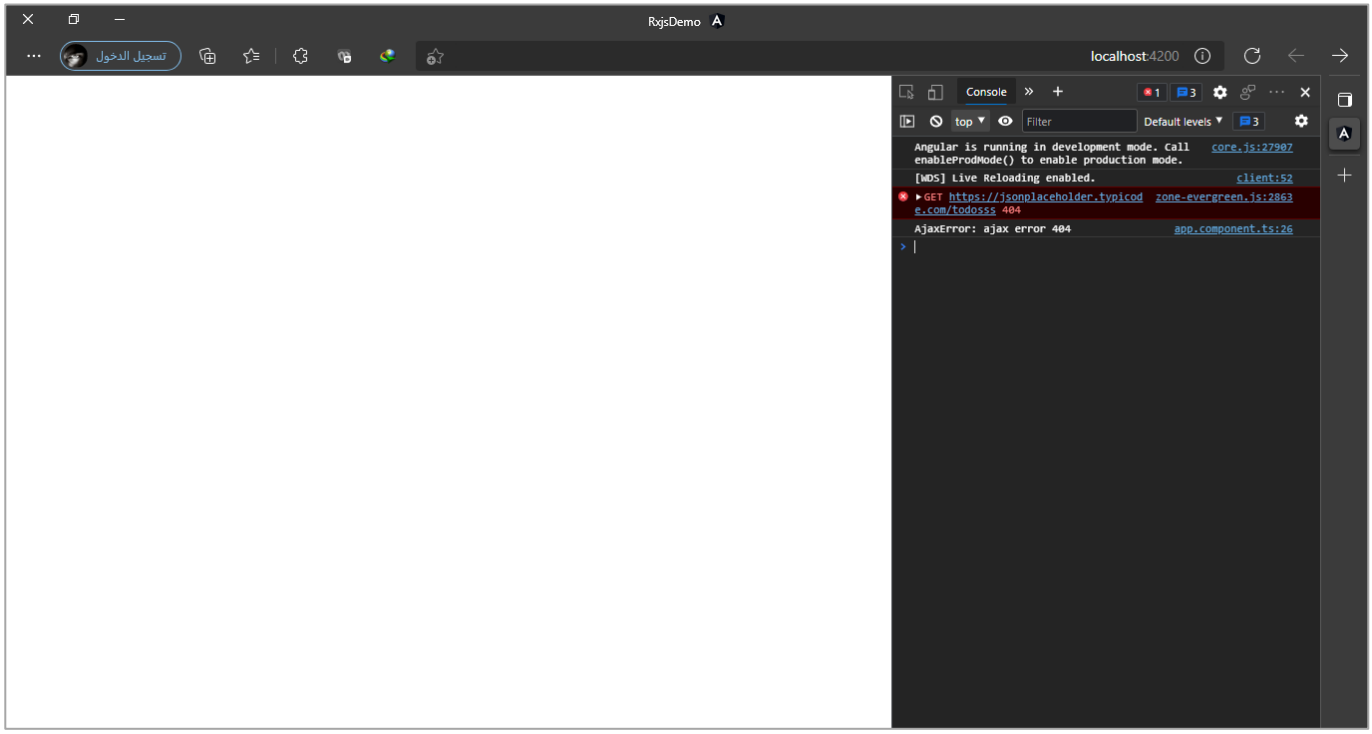
```
import { Component, OnInit } from '@angular/core';
import { catchError } from 'rxjs/operators';
import { ajax } from 'rxjs/ajax';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    ajax
      .getJSON('https://jsonplaceholder.typicode.com/todosss')
      .pipe(
        catchError((error, obs) => {
          if (error.name === 'AjaxError') {
            throw new Error(error);
          } else {
            throw new Error('Unknown Error');
          }
        })
      )
      .subscribe({
        next: (value: any) => console.log(value),
        error: (error) => console.log(error.message || error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

في المثال بالأعلى قمنا بعمل اتصال بقاعدة بيانات عن طريق API معين بواسطة تقنية AJAX، ومن ثم قمنا بعمل خطأ مقصود، وذلك عن طريق تغيير في الرابط حيث اضفنا ss في آخر الرابط، ومن ثم قمنا بمعالجة هذا الخطأ عن طريق الدالة catchError في الدالة pipe أي قبل لا يصل إلى الSubscription وهذه الدالة تلتقط الخطأ بشكل مباشر وتقوم بتخزينه في اول باراميتر يُمرر لها وفي حالتنا هنا اسميته error (لك عزيزي المتعلم اختيار أي اسم تُريده) اما الباراميتر الثاني والذي اسميته obs فيمثل الObservable الأساسي الذي وقع فيه الخطأ.

وكما قلنا ان هذه الدالة تتيح لنا ان نلتقط هذا الخطأ ومن ثم نعيده او نُعيد أي خطأ آخر بحسب احتياجنا وهو ما قمنا بعمله هنا حيث مررنا هذا الخطأ على شرط بحيث إذا كان الخطأ هو من النوع AjaxError (احد الأنواع التي تقدمها لنا مكتبة rxjs وهي خارج نطاق هذا الكتاب لأننا في الAngular نستخدم مديول جاهز يقدم لنا خيارات كثيرة لتعامل مع API والأخطاء التي قد تقع منها) فإذا كان من النوع AjaxError قم برمي نفس الخطأ اما إذا كان أي خطأ آخر فقم برمي رسالة نصية تحمل صيغة خطأ معينة.

وأخيراً استقبلنا الخطأ في Subscription وعرضناه للمستخدم، بحيث تكون النتيجة كالتالي:



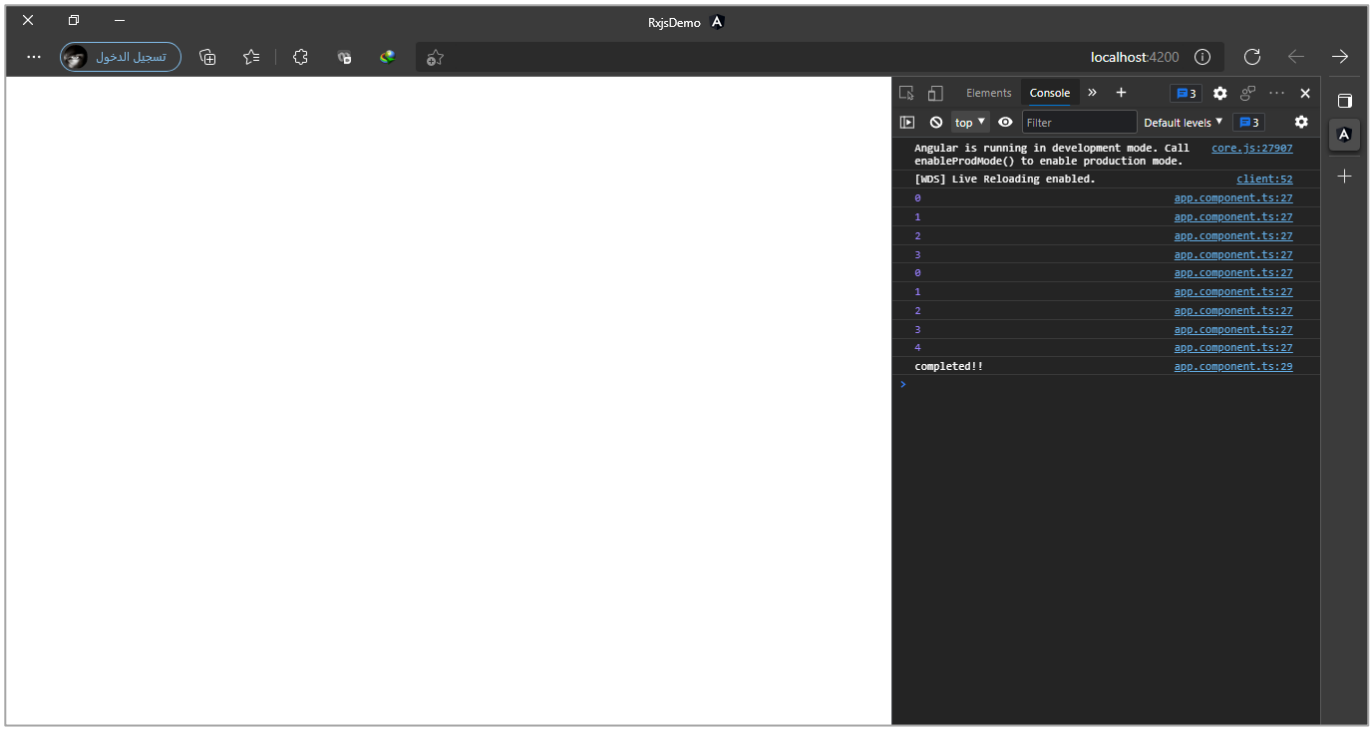
والآن لنُعطي مثال آخر على كيفية التقاط الخطأ وإعادة Observable آخر، مهما كان هذا الObservable بحسب الاحتياج، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { catchError, map, take } from 'rxjs/operators';
import { interval } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    interval(1000)
      .pipe(
        map((value) => {
          if (value >= 4) {
            throw new Error('Error Happened');
          }
          return value;
        }),
        catchError((error, obs) => {
          return interval(1000).pipe(take(5));
        })
      )
      .subscribe({
        next: (value: any) => console.log(value),
        error: (error) => console.log(error),
```

```
        complete: () => console.log('completed!!'),
      });
  }
}
```

كما هو واضح في المثال السابق قمنا بعمل خطأ وهو في حالة كانت القيم اكبر من 4، ومن ثم التقطنا هذا الخطأ عن طريق الدالة catchError ولكن بدلاً من التعامل مع الخطأ نفسه قمنا بتجاهل هذا الخطأ وأعدنا Observable جديد، مهما يكن هذا الObservable بحسب احتياجك عزيزي المتعلم، وستكون النتيجة كالتالي:



كما هو ملاحظ تم عرض اول اربع قيم ومن ثم تم رمي الخطأ ولكن هنا تجاهلنا الخطأ واعدنا Observable وفي حالتنا اعدنا الدالة interval واخذنا اول خمس قيم عن طريق الدالة take لأننا لا نُريد ان يستمر العد إلا مالا نهاية.

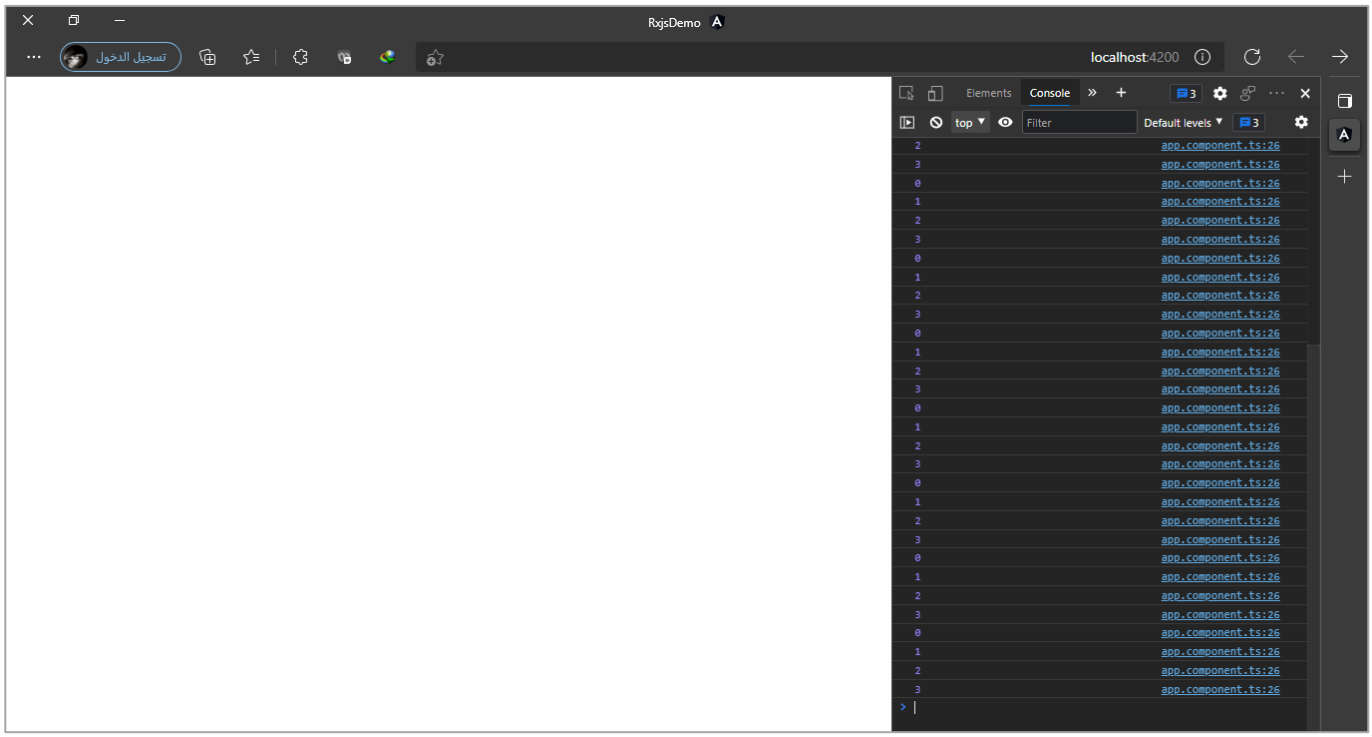
كما نستطيع ان نُعيد الObservable نفسه ولكن هنا لابد ان ننتبه اننا سنخدل في حلقة تكرار لا نهائية infinite Loop بحيث سيقرأ اول اربع قيم وسيتم رمي الخطأ ومن ثم نلتقطه في الدالة catchError وعن طريق هذه الدالة سنُعيد نفس الObservable لذلك سيقوم بإعادة قراءة اول اربع قيم وبعده سيتم رمي الخطأ ويتم التقاطه بواسطة الدالة catchError وهكذا ستتم هذه العملية إلا مالا نهاية، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { catchError, map } from 'rxjs/operators';
import { interval } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

```
})

export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    interval(1000)
      .pipe(
        map((value) => {
          if (value >= 4) {
            throw new Error('Error Happened');
          }
          return value;
        }),
        catchError((error, obs) => {
          return obs;
        })
      )
      .subscribe({
        next: (value: any) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

والنتيجة ستكون:



7-3- Conditional and Boolean Operators:

هذه الدوال تقوم بالمقارنة بين قيم Observable واحد او بين اثنين او اكثر من الObservables وفق شرط او شروط معينة وتُعيد قيمة منطقية من النوع Observable أي انها تُعيد true إذا كانت المقارنة صحيحة والشرط تحقق وتُعيد false إذا حصل العكس، وكلا true/false يكونا على شكل Observable، ما عدا دالة واحدة نمرر لها قيمة افتراضية في حالة كان فارغ.

مع العلم اننا استعرضنا دالة تقوم بنفس فكرة هذه الدوال وهي دالة iif وهي أحدى دوال static حيث تُقارن بين قيم اثنين Observables وتعيد قيمة منطقية، اما هنا فستكون على مستوى دوال Pipe مع بعض الدوال الأخرى والتي سوف نستعرضها، كالتالي:

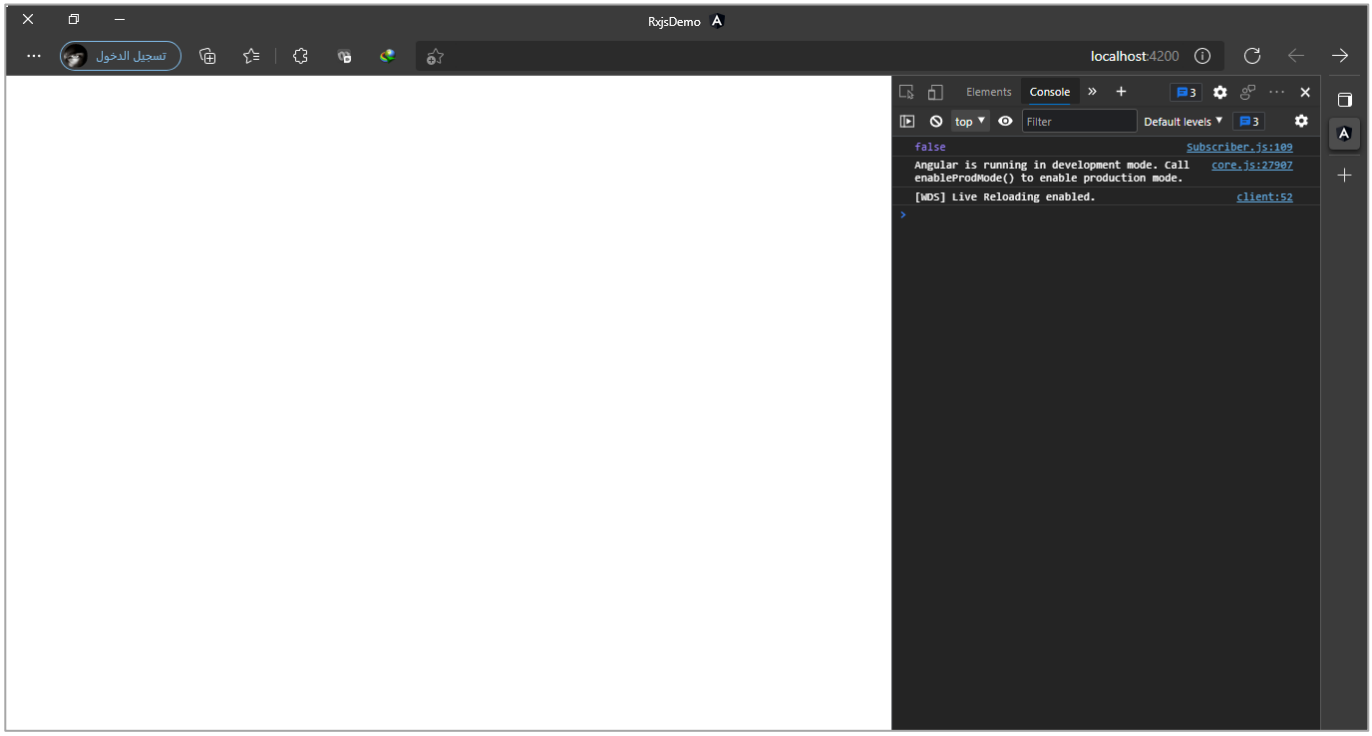
7-3-1- every():

هذه الدالة تتعامل مع قيم Observable واحد حيث نمرر لها شرط معين وفي حال تحقق هذا الشرط على جميع القيم تُعيد true من النوع Observable وتُعيد false إذا حدث العكس، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { every } from 'rxjs/operators';
import { from, fromEvent, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(0, 2, 3, 4, 6)
      .pipe(
        every((val: number) => val % 2 === 0)
      )
      .subscribe(console.log);
  }
}
```

كما نلاحظ يوجد لدينا Observable يحتوي على مجموعة من القيم الرقمية ومررنا هذه القيم على الدالة every ووضعنا لها شرط بحيث نتأكد هل القيم والأعداد زوجية ام لا؟ ومن ثم نطبع الناتج في console وكما ذكرنا سابقاً سوف يطبع لنا اما true او false، ولنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ تم طباعة false وذلك لأن القيمة 3 قيمة فردية ولم تحقق الشرط وهذا الذي قلناه سابقاً وهو لابد جميع القيم تحقق الشرط المُعطى لكي تُعيد true.

3-7-2- sequenceEqual():

اما هذه الدالة فتقوم بالمقارنة بين قيم اثنين Observables بحيث يكون احد الObservables هو الأساس الذي نُريد ان نقارنه بقيم الObservable الآخر، وتختلف عن الدالة السابقة انها تقارن كل قيمة على حدا ومن ثم تُعيد نتيجة هذه المقارنة true او false.

ومن هذا المنطلق نقول ان هذه الدالة نمرر لها Observable والذي اسميناه مجازاً الأساس ومن ثم تتكفل هي بمقارنة هذا الObservable المرر لها مع الObservable الآخر وتعيد نتيجة كل مقارنة على حدا.

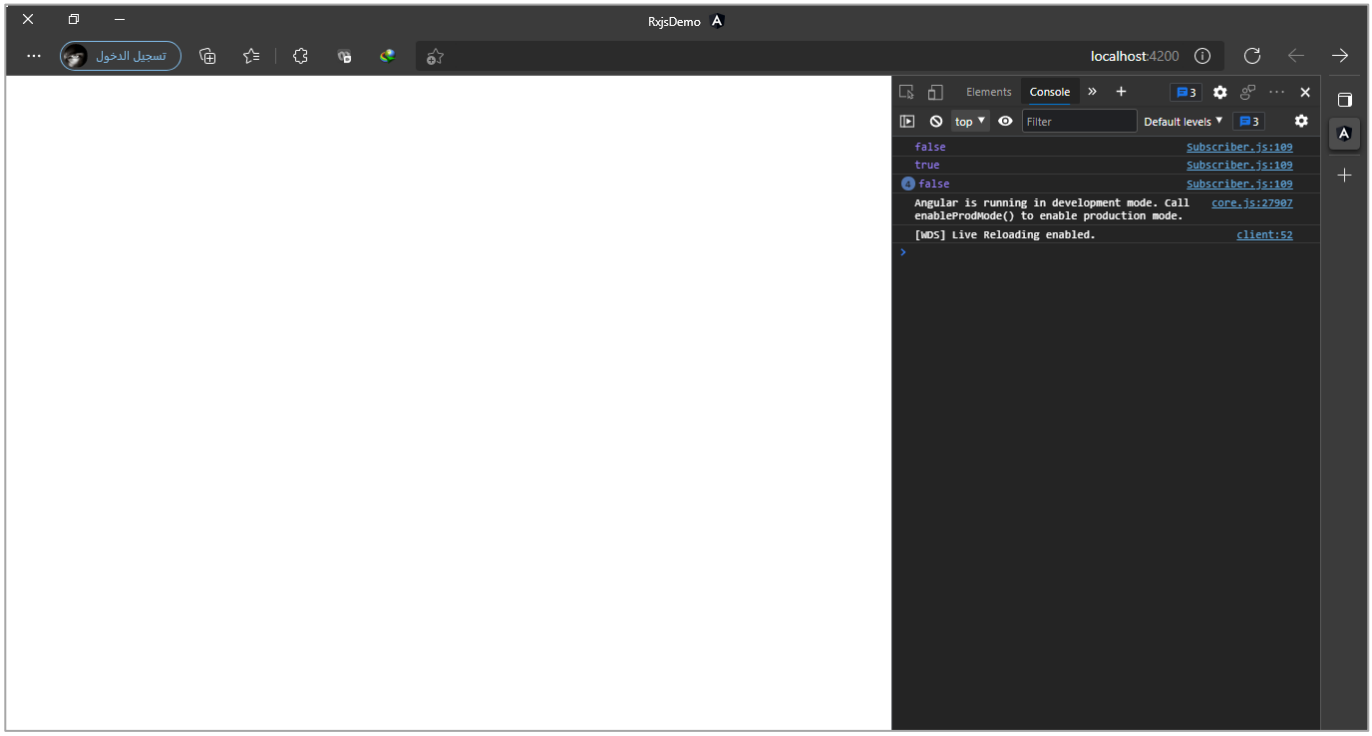
ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { sequenceEqual, switchMap } from 'rxjs/operators';
import { of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor({}
  ngOnInit(): void {
    const four = of(4);
```

```
    of(1, 4, 6, 5, 7, 8)
      .pipe(switchMap((values) => of(values).pipe(sequenceEqual(four))))
      .subscribe(console.log);
  }
}
```

كما هو واضح من المثال السابق قمنا بتعريف Observable باسم four ويحمل قيمة واحدة وهي الرقم 4 بحيث يكون هو الأساس، ومن ثم قمنا بتعريف Observable آخر يحتوي على مجموعة من القيم، ومن ثم نُريد ان نُقارن بين هذا الأساس وقيم الObservable الآخر لذلك استخدمنا الدالة sequenceEqual ومررنا لها الObservable الذي اسميناه four، اما الآن لنشاهد النتيجة، كالتالي:



كما نلاحظ قارن بين كل قيمة في الObservable الآخر مع قيمة الObservable الأساس وقام بإعادة نتيجة كل قيمة على حدا.

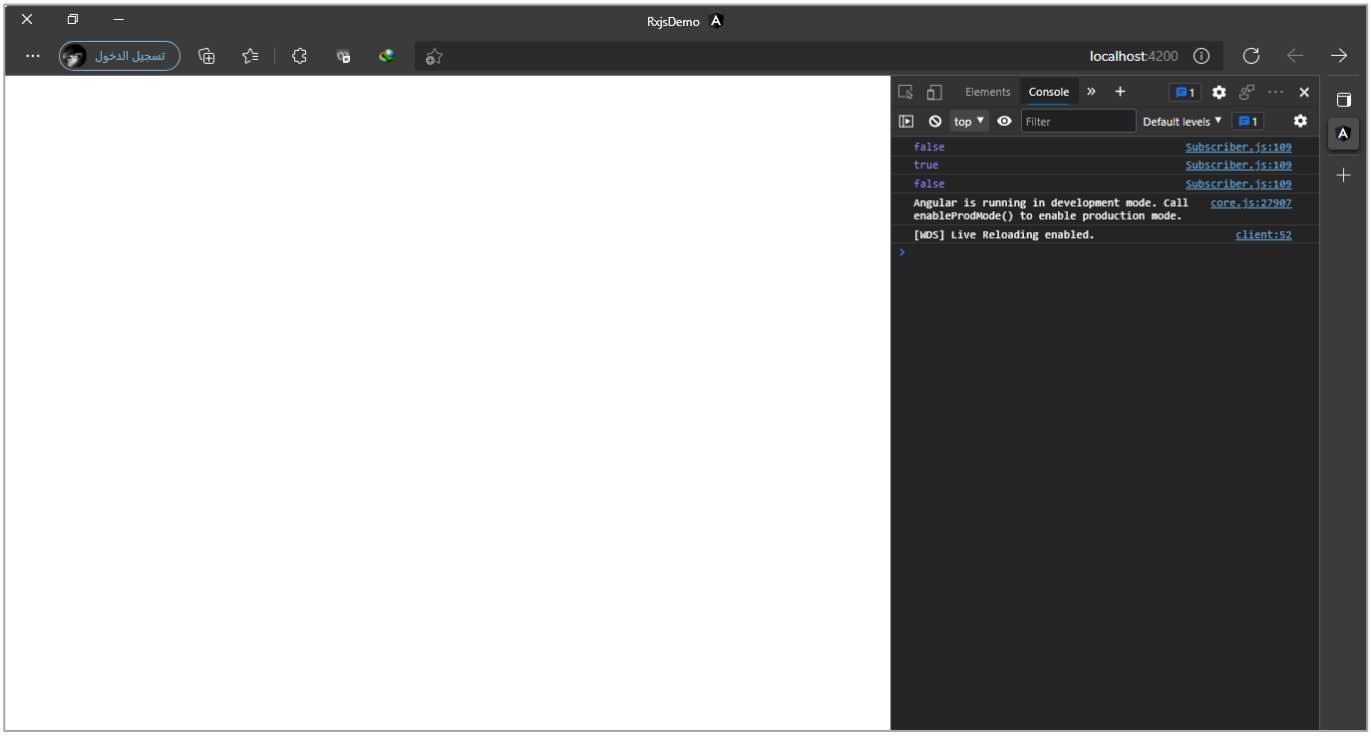
كما اننا نستطيع ان نُقارن بين مجموعة قيم مع مجموعة قيم وليس فقط قيمة واحدة، كأن نُقارن مصفوفة مع مجموعة من المصفوفات ونعيد نتيجة كل مقارنة على حدا، كما في المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { sequenceEqual, switchMap } from 'rxjs/operators';
import { of, from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor({}
  ngOnInit(): void {
    const sequence = from([1, 2]);
    of([4, 5], [1, 2], [6, 7])
      .pipe(switchMap((values) => from(values).pipe(sequenceEqual(sequence))))
      .subscribe(console.log);
  }
}
```

وتكون النتيجة، كالتالي:



ولنُعطي آخر مثال لهذه الدالة ومصدره (sequenceEqual - Learn RxJS) مع بعض التعديلات البسيطة التي قمت بها عليه. وهذا المثال يُراجع لنا مجموعة من الدوال السابقة التي قمنا بالتطرق لها وايضاً بعض التقنيات الأخرى التي تطرقت لها في الكتب السابقة من هذه السلسلة.

وفكرة هذا المثال بكل بساطة هو لدينا كلمة مقسمة حروفها على شكل سيل من قيم الObservable ومن ثم نقارن بين هذه القيم وبين ما يكتبه المستخدم في لوحة المفاتيح وبناءً عليه يصبح الObservable الأول هو الأساس ونقارنه بالObservable الثاني وهو ضغطات المستخدم على لوحة المفاتيح، كالتالي:

ملف app.component.html

```
<div>
  <p class="result" #result></p>
  <p #msg></p>
</div>
```

ملف app.component.scss

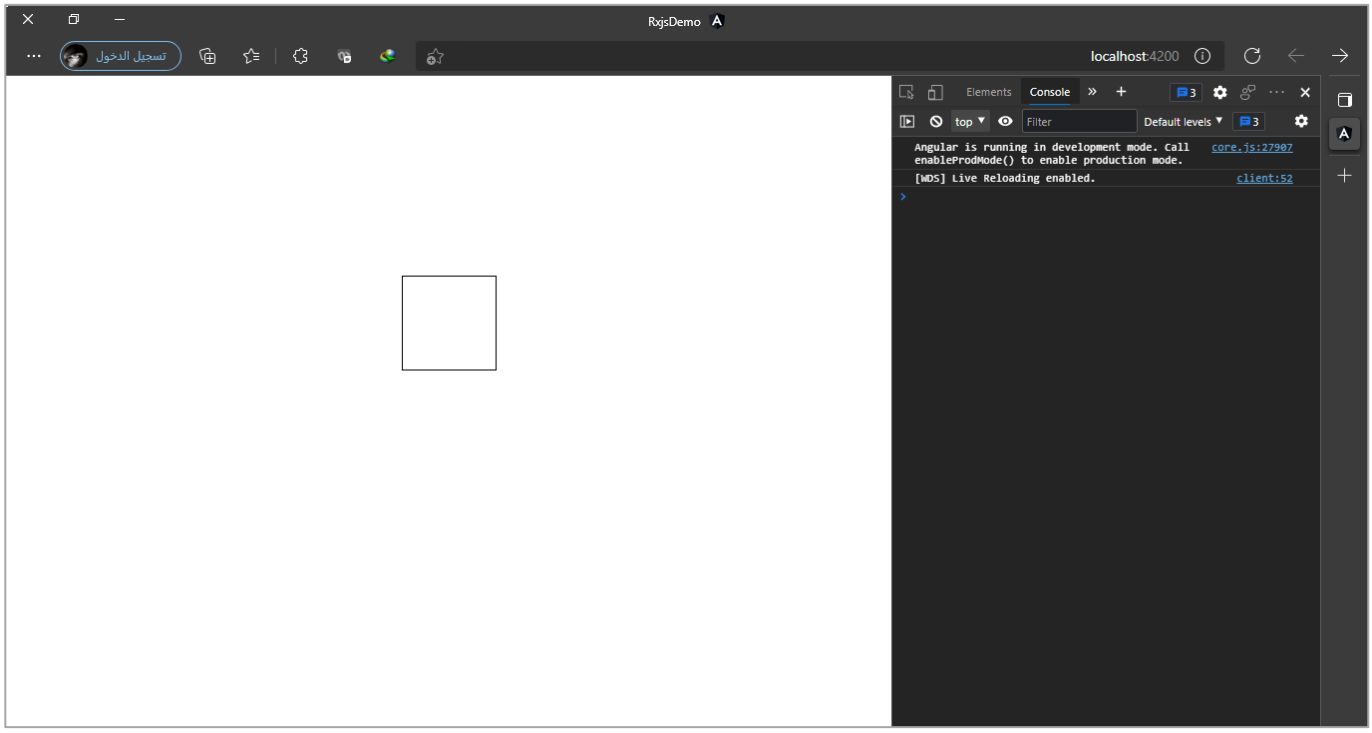
```scss
div {
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 80vh;
}
.result {
  box-sizing: border-box;
  width: 100px;
  height: 100px;
  text-align: center;
  font-weight: bolder;
  font-size: larger;
  border: 1px solid black;
  padding: 20px;
}
```

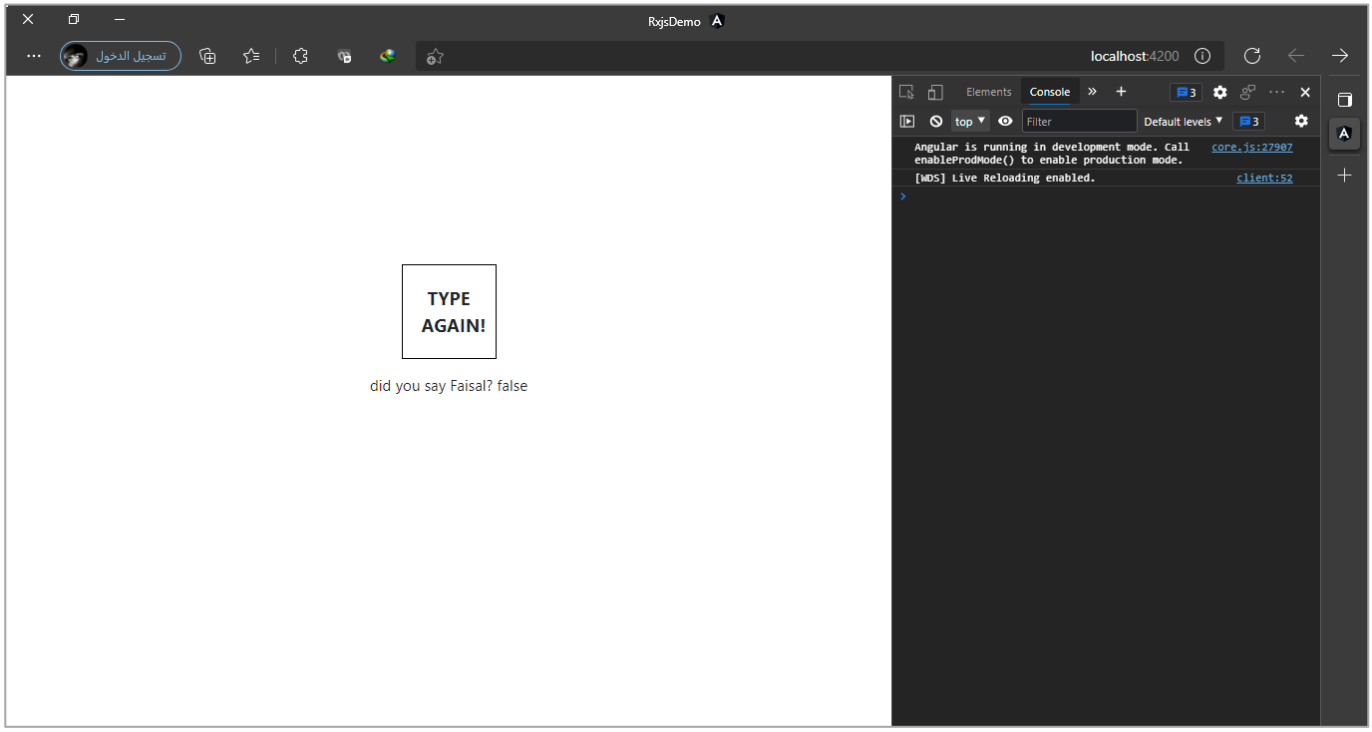
ملف app.component.ts

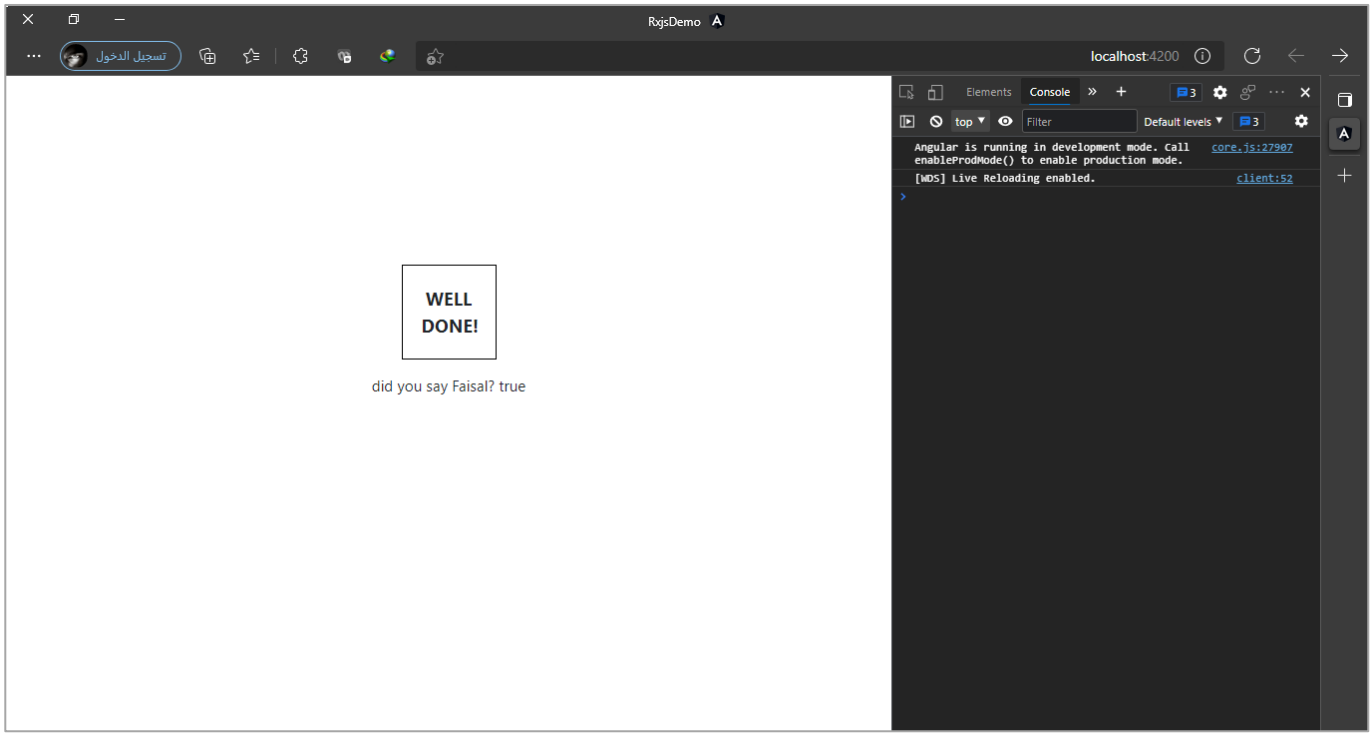
```ts
import { Component, ElementRef, OnInit, Renderer2, ViewChild } from '@angular/core';
import { bufferCount, map, mergeMap, sequenceEqual, tap } from 'rxjs/operators';
import { from, fromEvent, of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor(private renderer: Renderer2) {}

  @ViewChild('result') private result: ElementRef<HTMLParagraphElement>;
  @ViewChild('msg') private msg: ElementRef<HTMLParagraphElement>;

  ngOnInit(): void {
    const expectedSequence = from(['f', 'a', 'i', 's', 'a', 'l']);
    fromEvent(document, 'keydown')
      .pipe(
        map((e: KeyboardEvent) => e.key),
        tap((value) => {
          this.renderer.setProperty(
            this.result.nativeElement,
            'innerText',
            value
          );
          this.renderer.setProperty(this.msg.nativeElement, 'innerText', '');
        }),
        bufferCount(6),
        mergeMap((keyDowns) =>
```

```
          from(keyDowns).pipe(
            sequenceEqual(expectedSequence),
            tap((isHeFaisal) => {
              const text = isHeFaisal ? 'WELL DONE!' : 'TYPE AGAIN!';
              this.renderer.setProperty(
                this.result.nativeElement,
                'innerText',
                text
              );
            })
          )
        )
      )
      .subscribe((e) => {
        this.renderer.setProperty(
          this.msg.nativeElement,
          'innerText',
          `did you say Faisal? ${e}`
        );
      });
  }
}
```

ولنشاهد النتيجة في المتصفح، كالتالي:







3-7-3- defaultIfEmpty():

هذه الدالة فكرتها بسيطة حيث ترتكز مهمتها الأساسية على تمرير قيمة معينة في حالة كان الObservable فارغ.

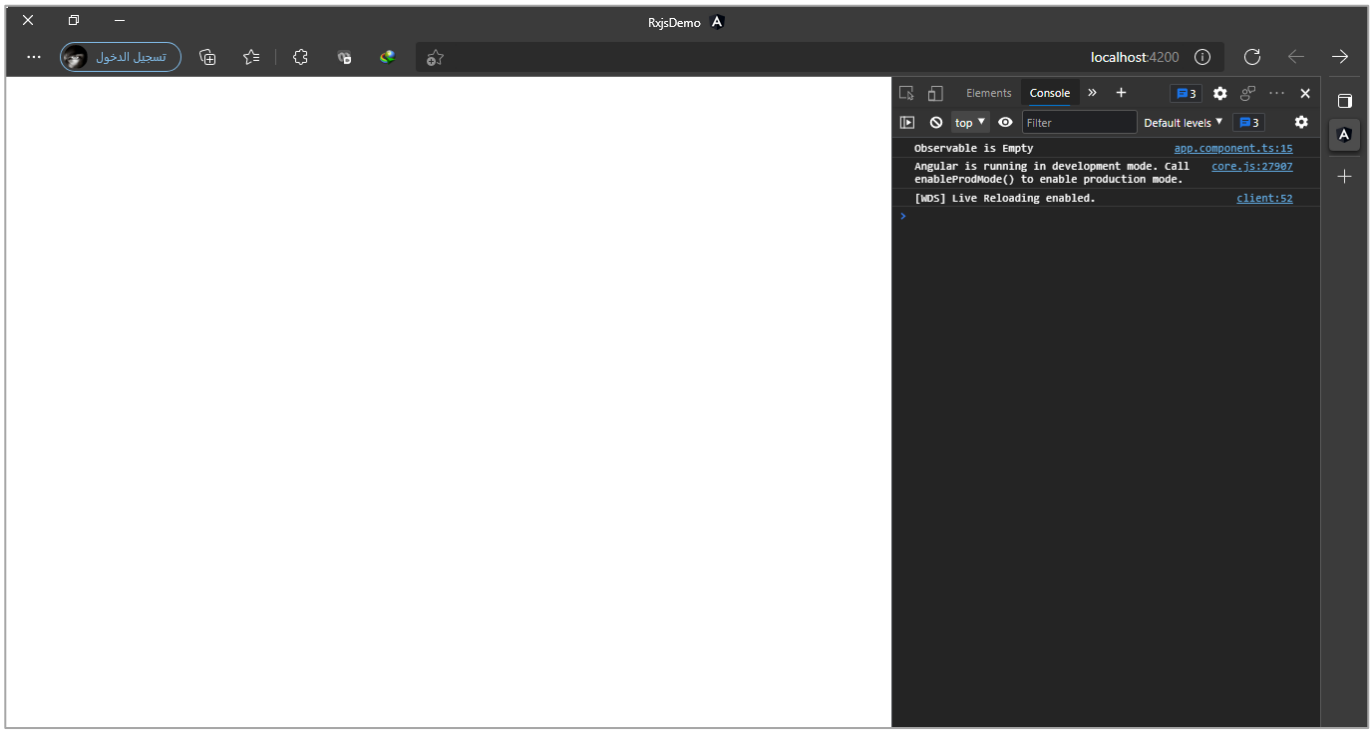
ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { defaultIfEmpty } from 'rxjs/operators';
import { of } from 'rxjs';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of()
      .pipe(defaultIfEmpty('Observable is Empty'))
      .subscribe((val) => console.log(val));
  }
}
```

كما نلاحظ في حالة كان الObservable فارغ مررنا قيمة نصية، مع العلم اننا نستطيع تمرير أي قيمة نُريدها.



8-3- Mathematical and Aggregate Operators:

هذه الدوال تقوم بإجراء بعض العمليات الحسابية على الObservable حيث نستطيع ان نوجد اكبر قيمة عن طريق الدالة max او اصغر قيمة باستخدام الدالة min او عدد القيم بواسطة الدالة count او حتى مجموع القيم باستخدام الدالة التي تكلمنا عنها سابقاً وهي دالة reduce او الدالة scan وبما اننا تكلمنا عن هاتين الدالتين سابقاً فلن نستعرضهم هنا والسبب الذي دفعني لوضعهما في مجموعة Transformation Operators وليس مع Mathematical Operators هو ان الاستخدام الشائع لهما هو في التحويل والتلاعب في البيانات وليس لإجراء عملية الجمع الرياضية.

اما الآن لنستعرض الدوال الثلاثة في هذه المجموعة، كالتالي:

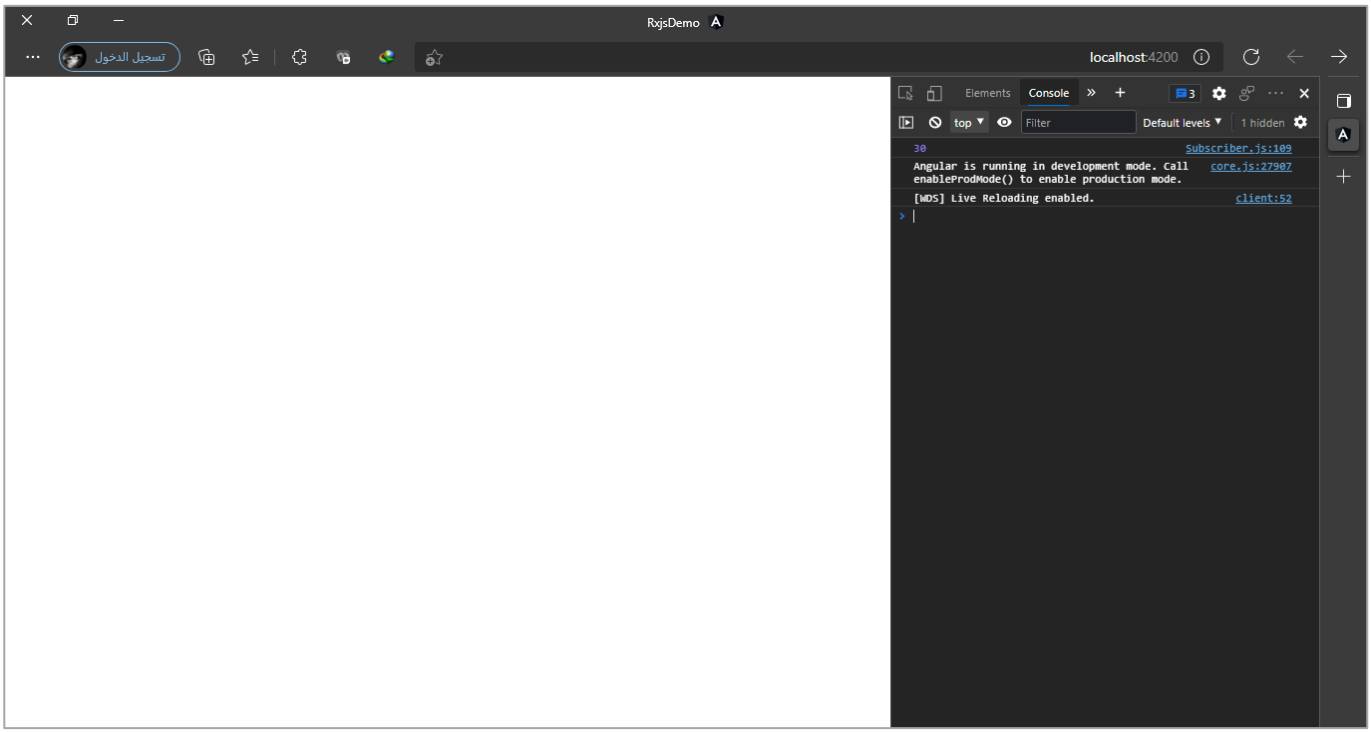
3-8-1- max() :

هذه الدالة تقوم بإيجاد أكبر قيمة في الObservable سواء كانت القيم بسيطة او معقدة كأن تكون قيمة رقمية جزء من كائن، وسوف نستعرض هذه الحالات من خلال الأمثلة كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { max } from 'rxjs/operators';
import { of } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(30, 1, 7, 0, 10).pipe(max()).subscribe(console.log);
  }
}
```

كما نلاحظ لدينا مجموعة من القيم ومررنا هذه القيم على الدالة max بدون تمرير أي باراميتر، والآن لنشاهد النتيجة كالتالي:



كما نلاحظ تم عرض اعلى قيمة وهي 30.

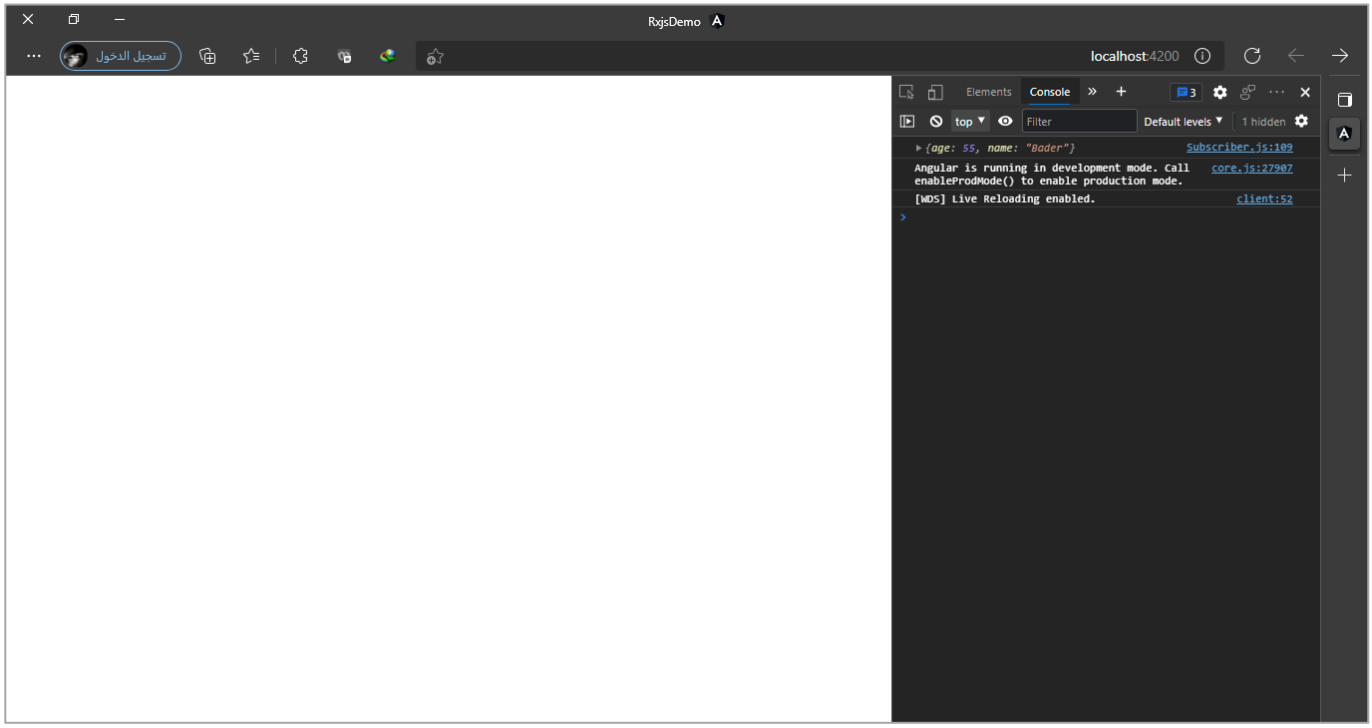
والآن لنستعرض بيانات أكثر تعقيداً وهو عبارة عن مجموعة من الكائنات وتحمل مجموعة من القيم، بحيث نأخذ اكبر قيمة.

ولتوضيح لنفرض لدينا مجموعة من الكائنات وكل كائن يحتوي على قيمتين العمر والاسم ونُريد ان نأخذ اكبر قيمة للعمر من جميع الكائنات، ونستطيع القيام بهذا الأمر عن طريق تمرير دالة Function وهذه الدالة تستقبل بارامترين بحيث البارامتر الأول يمثل الكائن الأول والبارامتر الثاني يمثل البارامتر الذي يليه وهكذا يتم الانتقال بين الكائنات إلى ان يتم الانتهاء من جميع قيم Observable وفي كل مرة نقوم بإجراء عملية مقارنة بين قيم العمر ونُعيد قيمة رقمية 1 تمثل true أي ان المقارنة صحيحة و1- تمثل false أي المقارنة غير صحيحة، كما في المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { max } from 'rxjs/operators';
import { of } from 'rxjs';

export interface Person {
  age: number;
  name: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(
      { age: 22, name: 'Fahd' },
      { age: 55, name: 'Bader' },
      { age: 24, name: 'Saad' }
    )
      .pipe(
        max<Person>((x: Person, y: Person) => {
          return x.age > y.age ? 1 : -1;
        })
      )
      .subscribe(console.log);
  }
}
```

والنتيجة، كالتالي:



3-8-2- min():

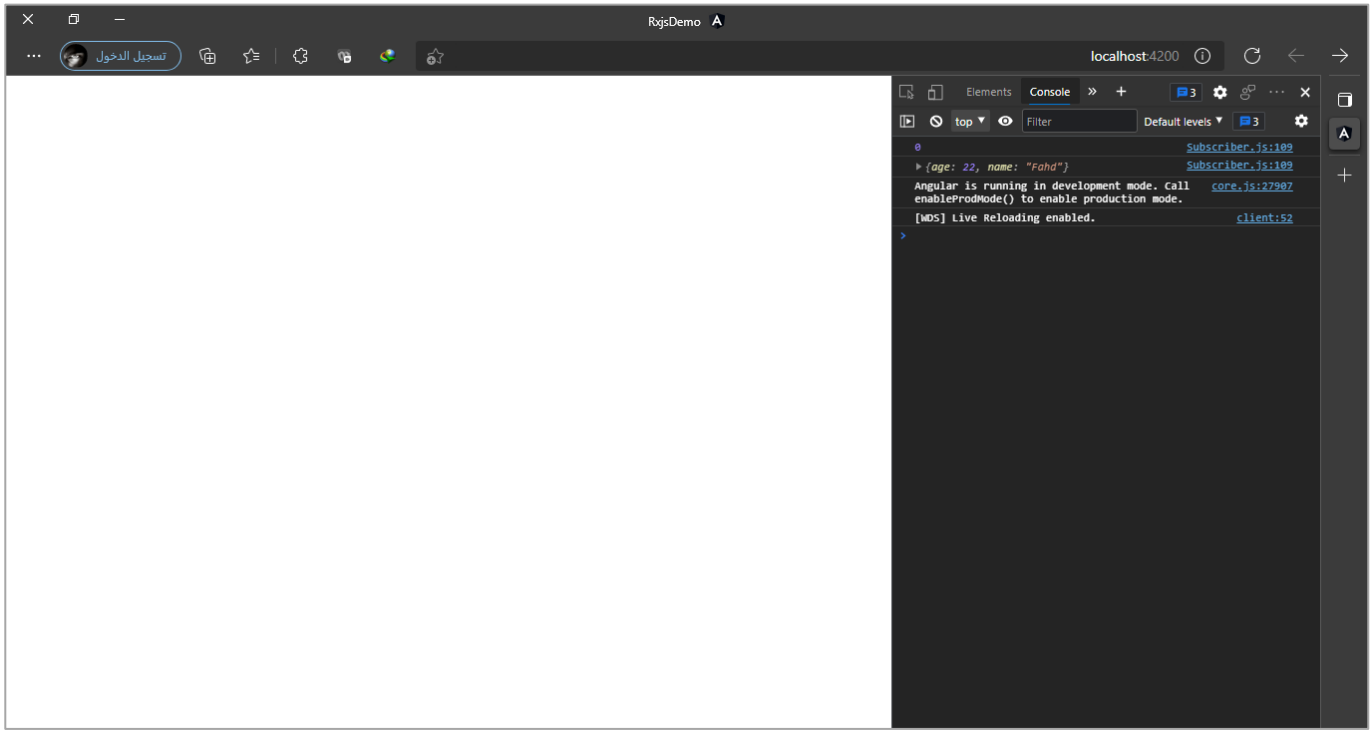
اما هذه الدالة فهي عكس الدالة السابقة بحيث تقوم بإيجاد أقل قيمة، لذلك سوف نقوم بتطبيق الأمثلة السابقة عليها، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { min } from 'rxjs/operators';
import { of } from 'rxjs';

export interface Person {
  age: number;
  name: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(30, 1, 9, 0, 14).pipe(min()).subscribe(console.log);
    of(
      { age: 22, name: 'Fahd' },
      { age: 55, name: 'Bader' },
      { age: 24, name: 'Saad' }
    )
      .pipe(
        min<Person>((x: Person, y: Person) => {
```

```
          return x.age > y.age ? 1 : -1;
        })
      )
      .subscribe(console.log);
  }
}
```

والنتيجة:



3-8-3- count():

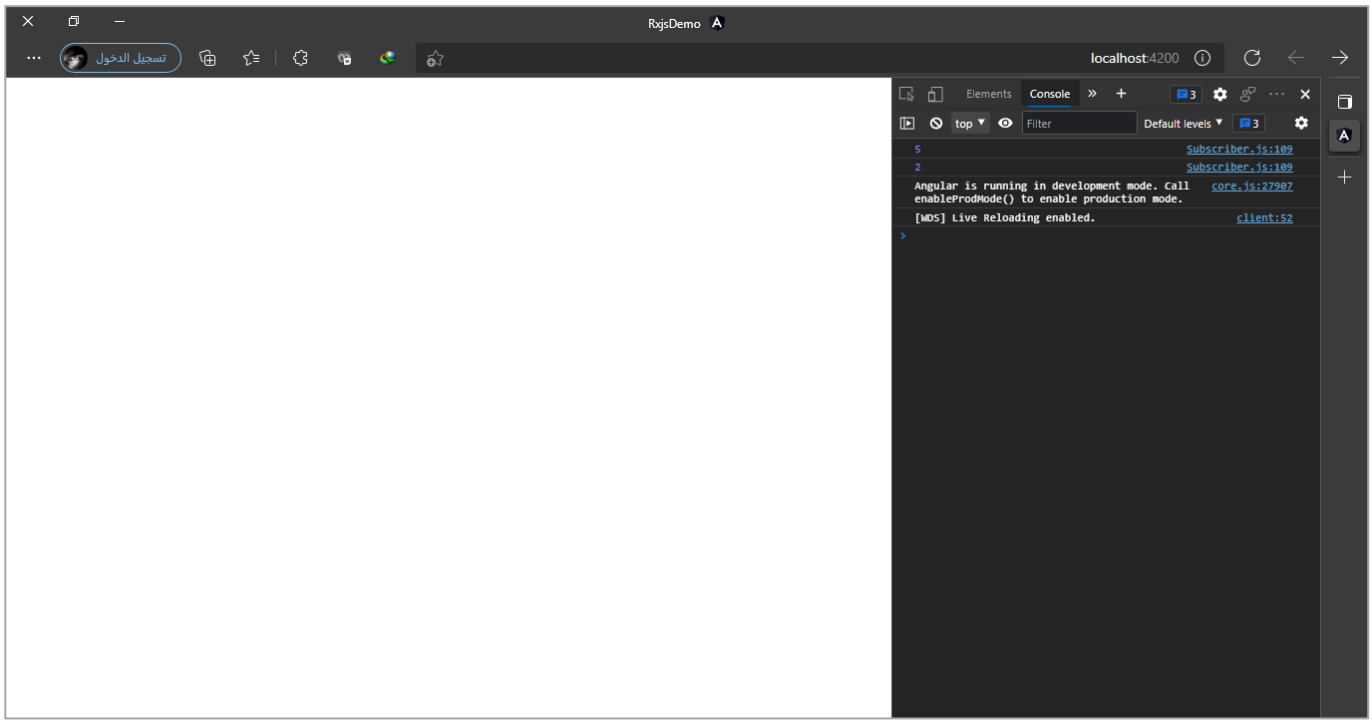
اما هذه الدالة فتقوم بعرض عدد القيم في الObservable، وليس هذا فحسب وانما نستطيع ايضاً ان نستعرض عدد القيم التي تحقق شرط معين، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { count } from 'rxjs/operators';
import { of } from 'rxjs';

export interface Person {
  age: number;
  name: string;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor() {}
  ngOnInit(): void {
    of(30, 1, 9, 0, 14).pipe(count()).subscribe(console.log);
    of(
      { age: 22, name: 'Fahd' },
      { age: 55, name: 'Bader' },
      { age: 24, name: 'Saad' },
      { age: 40, name: 'Fahd' }
    )
      .pipe(
        count((value: Person, index: number) => {
          return value.name === 'Fahd' ? true : false;
        })
      )
      .subscribe(console.log);
  }
}
```

في الجزء الأول قمنا بتطبيق الدالة count لمعرفة عدد جميع القيم في الObservable، اما الثاني فنقوم بإرجاع عدد القيم التي تحقق الشرط وهو الكائنات التي خاصية name تحمل القيمة Fahd، بحيث تكون النتيجة:



وهذا هو المتوقع حيث تم اظهار في البداية القيمة 5 وهو عدد قيم الObservable الأول وايضاً تم اظهار القيمة 2 وهو عدد القيمة التي حققت الشرط من الObservable الثاني.

الفصل الرابع

Angular HttpClient

1- مقدمة:

من المميزات المهمة التي تقدمها لنا الـAngular هي القدرة على التواصل مع أي سيرفر server من خلال بروتوكول Http/Https بغض النظر عن التقنية التي يستخدمها هذا السيرفر او لغة البرمجة التي يتبناها لتعامل مع قاعدة البيانات، وفي عالم الـJavaScript نستطيع التواصل مع السيرفر او بصيغة أخرى Back-end عن طريق تقنية AJAX لجلب وأرسال البيانات وايضاً المتصفح يتعامل مع طلبات Http عن طريق XMLHttpRequest او ما يسمى ايضاً fetch() API، وبما الـAngular هو بالأساس إطار عمل للغة الـJavaScript فلذلك قدمت لنا Angular تقنية – ميزة – مبنية على جميع التقنيات السابقة ولكن بطريقة سلسلة وسهلة ومميزات جداً رائعة وتتماشى مع أسلوب وطريقة الـAngular لجلب وارسال البيانات من السيرفر إلى العميل والعكس كذلك، وهذه الميزة تُسمى HttpClient.

ومن بعض المميزات التي تقدمها لنا هذه الميزة HttpClient ارسال طلب إلى السيرفر لقراءة البيانات او حفظها او حذفها او طلب للتعديل عليها، كما نستطيع التعديل على Headers سواء لإضافة بارامترات لتحديد صلاحيات مستخدم معين او لتحديد نوع البيانات التي يحتاجها تطبيقنا مثل xml, Json..الخ، وتقديمها ايضاً لطرق مختلفة لمعالجة الإخطاء، كما ان هذه الميزة HttpClient تُعيد Observable بشكل افتراضي، لذلك نستطيع الاستفادة من جميع مميزات ودوال وتقنيات مكتبة rxjs والتي تطرقنا لها سابقاً.

وقبل البدء في الغوص بهذه التقنية أحببت ان ابين جزء يُفترض بك عزيزي المتعلم ان تكون مُلم به ولكن لا ضير من المرور عليه لأننا سنستخدمه وبكثره خلال تعاملنا مع HttpClient، وهو Classes & Interfaces.

2- Classes & Interfaces in Typescript:

لن نغوص في تقنيات OOP وانما سنأخذ ما يخدمنا في ما سوف نشرحه في هذا الفصل، وفي الحقيقة وفي الغالب نستخدم هاتين التقنيتين في عمل وصف للبيانات القادمة لنا من الخادم لكي نساعد مُترجم Typescript لفهم هذه النوع الخاص بهذه البيانات، بمعنى آخر نستطيع ان نقول اننا باستخدام هاتين التقنيتين نقوم بإنشاء نوع جديد لوصف البيانات القادمة لنا من الخادم.

ولتوضيح لتعطي المثال التالي، لنفرض انه لدينا جدول في قاعدة البيانات يحتوي على مجموعة من البيانات، كالتالي:

User_Id	number
User_Name	string
address	string
Phone_Number	string
Age	Number
Salary	Number
Date_of_Birth_Day	date

فنحن كمطوري واجهات امامية نتوقع هذه البيانات من الخادم لذلك نقوم بإنشاء class او interface لوصف هذه البيانات او بالتحديد توليد نوع جديد لوصف هذه البيانات.

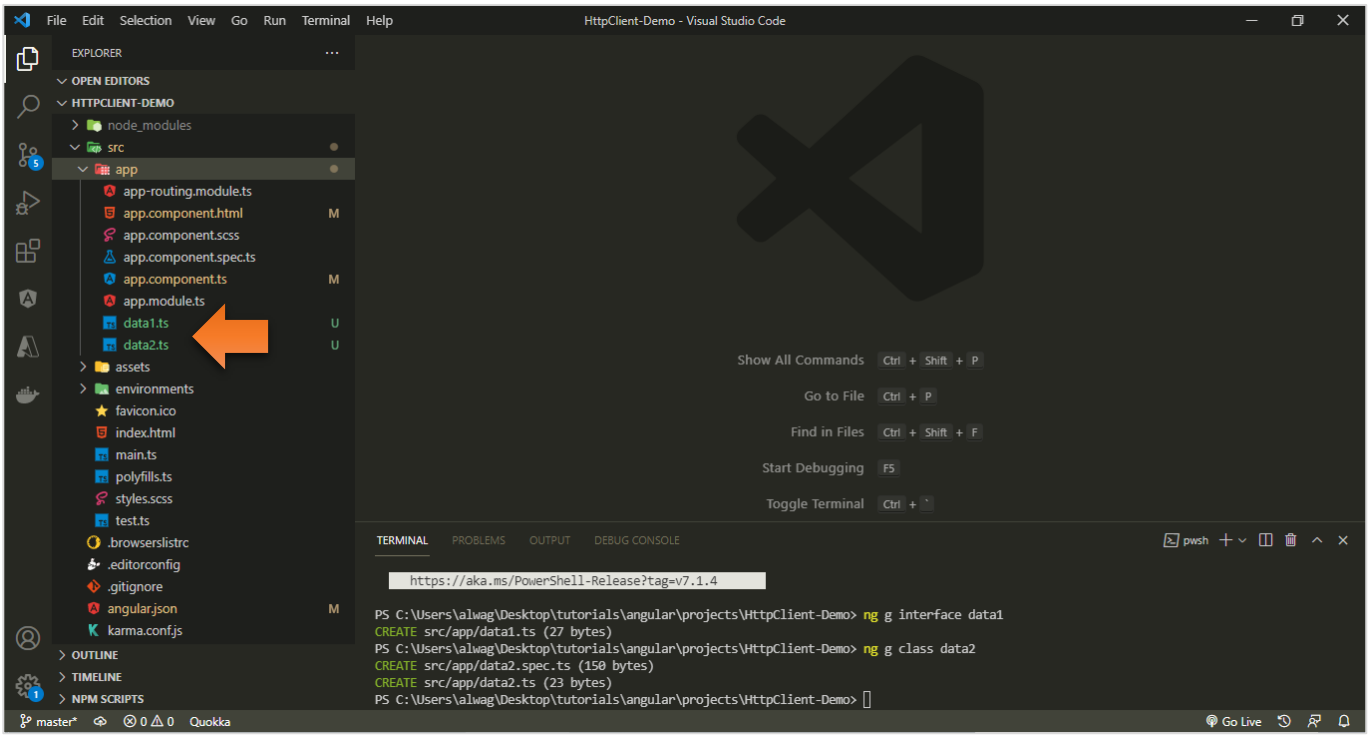
ولإنشاء interface او class نتأكد بأننا في نفس مسار المشروع ونكتب في terminal الامر التالي:

ng g interface data1

OR

ng g class data2

وبعد الانتهاء نقوم بكتابة



كما نلاحظ قمنا بإنشاء ملفين الأول عبارة عن interface ويحمل الاسم data1 والثاني عبارة عن class واسميته data2، والآن لذهب على ملف data1.ts ونكتب الوصف المناسب للبيانات القادمة لنا من الخادم، كالتالي:

ملف data1.ts

```
export interface Data1 {
  userId: number;
  userName: string;
  userAddress: string;
  userMobileNumber: number;
  userAge: number;
  userSalary: number;
  userDOB?: Date;
}
```

كما نلاحظ قمنا بوصف البيانات مع اتاحة الحرية باختيار أسماء المتغيرات التي نُريدها دون التقيد بنفس الأسماء القادمة من قاعدة البيانات، وايضاً نلاحظ انه يوجد علامة ? اما المتغير ذو الاسم userDOB وهذا معناه ان هذا المتغير غير الزامي وقد لا يحمل أي قيمة بعكس المتغيرات الثانية التي يجب ان تحمل قيمة.

ونفس الامر نعمله مع الملف الثاني data2.ts، كالتالي:

ملف data2.ts

```
export class Data2 {
  userId!: number;
  userName!: string;
  userAddress!: string;
  userMobileNumber!: number;
  userAge!: number;
  userSalary!: number;
  userDOB?: Date;
}
```

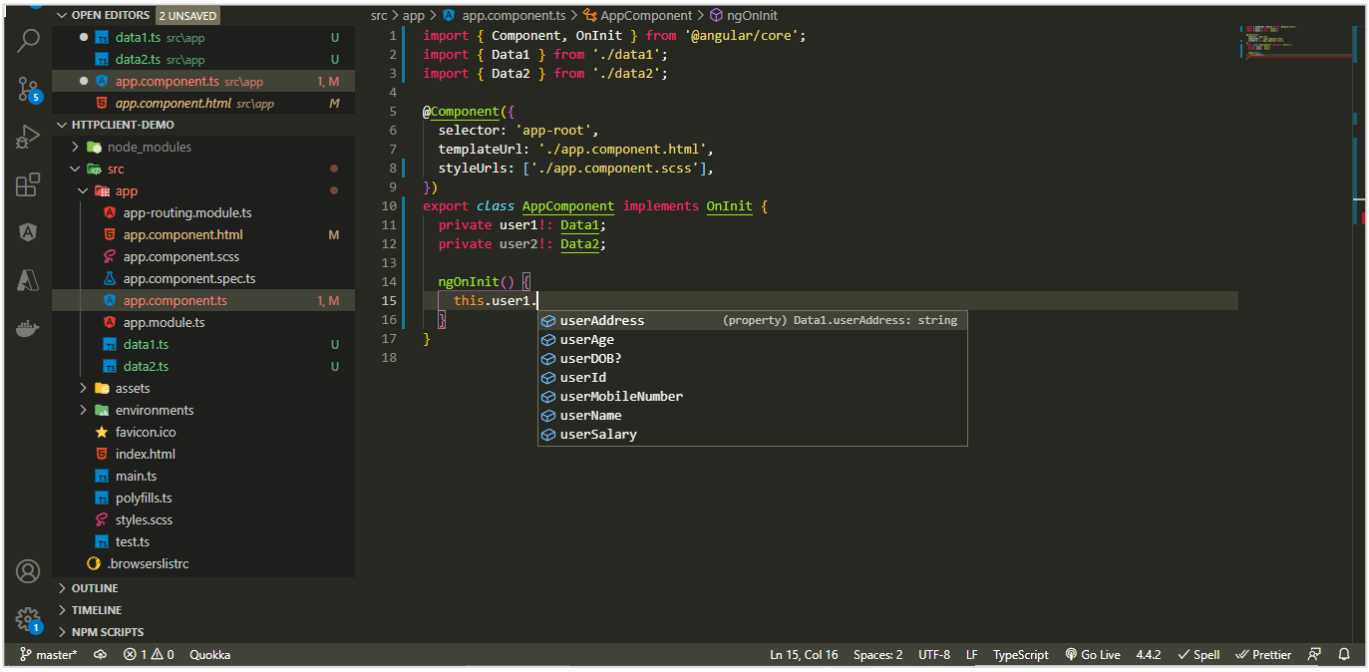
وهنا ايضاً قمنا بعمل وصف للبيانات ولكن هنا اضفنا العلامة ! امام المتغيرات ما عدا المتغير الأخير، وهذا الرمز لكيلا يحدث خطأ بأن لابد ان نقوم بعمل إعطاء قيمة مبدئية للمتغيرات مثلاً userId: number = 0، فهذا الرمز يغني عن هذا الامر، والأخير لا نحتاج ان نضع له الرمز ! لانه غير الزامي بوضعنا له الرمز ?.

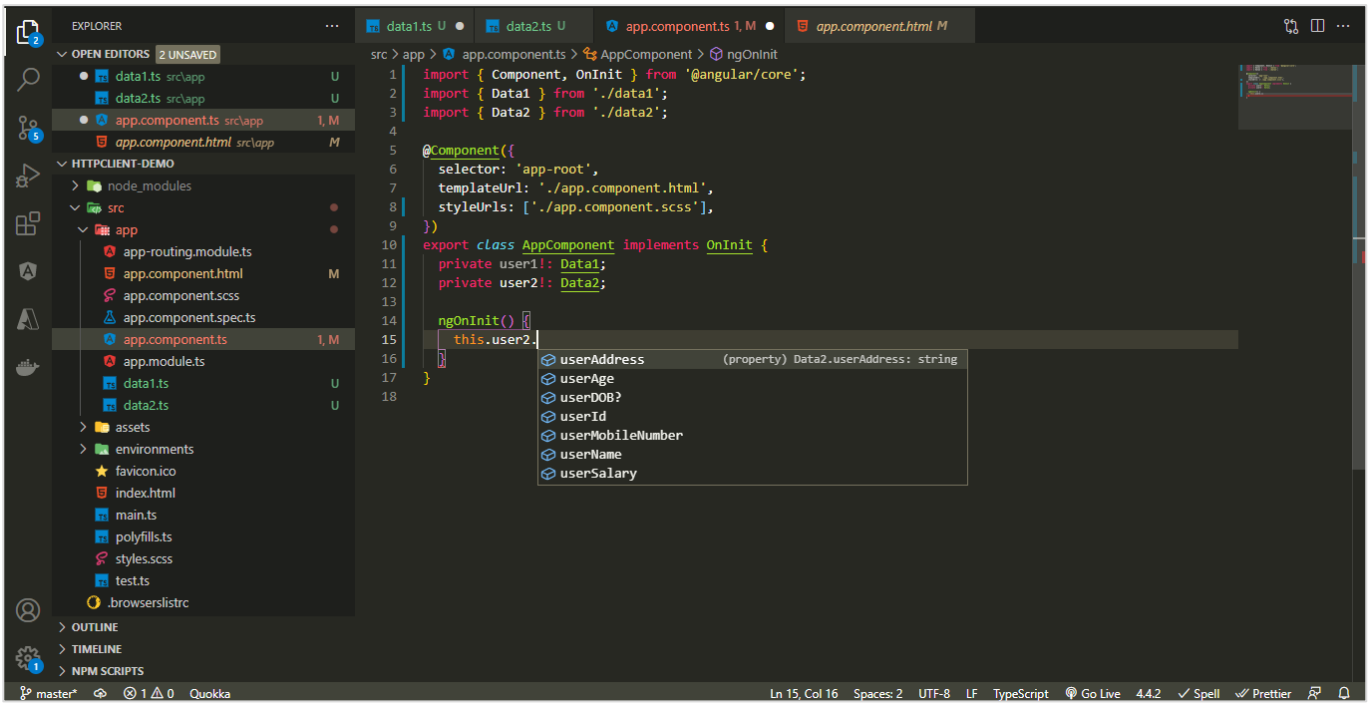
وبعد تعريفنا لهذه الأنواع لوصف البيانات نستطيع الآن استخدامها، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Data1 } from './data1';
import { Data2 } from './data2';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  private user1!: Data1;
  private user2!: Data2;

  ngOnInit() {}
}
```

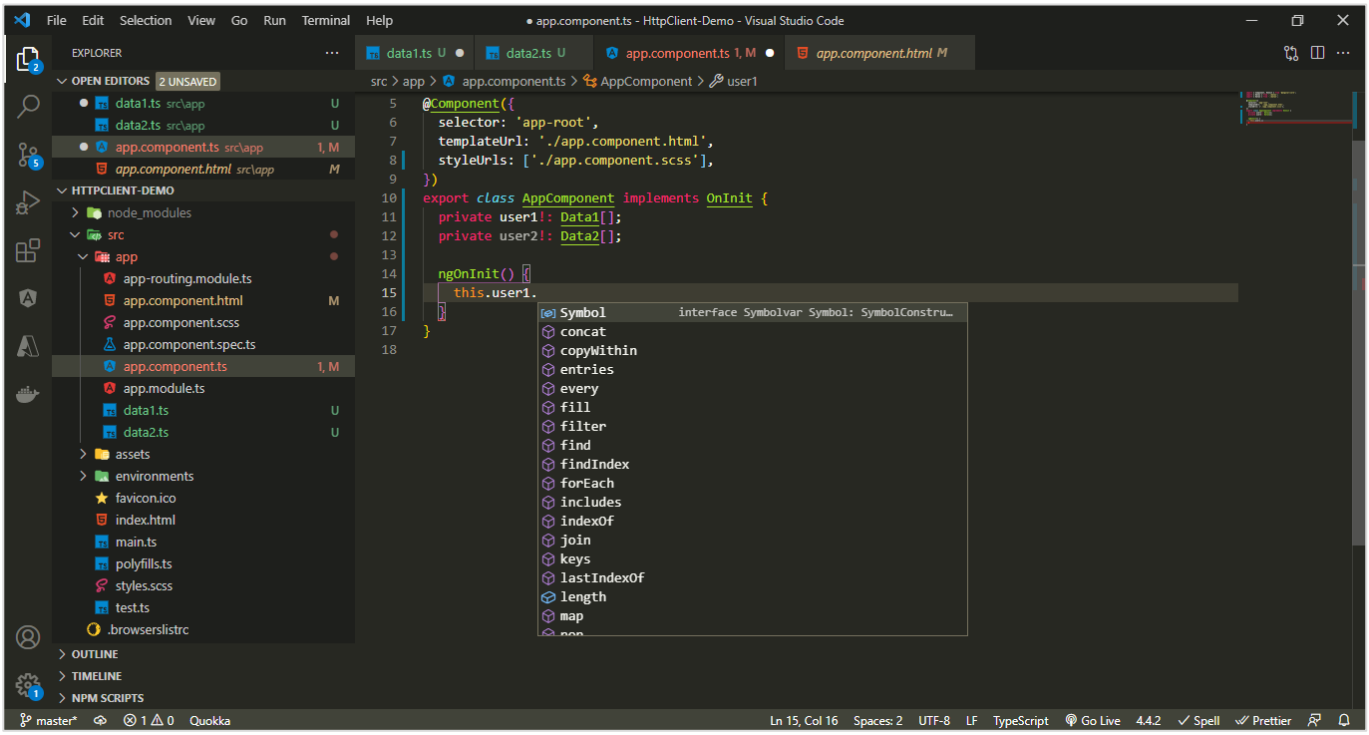
قمنا بتعريف متغيرين الأول باسم user1 وجعلنا نوعه Data1 والذي يمثل interface الذي قمنا بإنشائه سابقاً والثاني باسم user2 وجعلنا نوعه Data2 والذي يمثل class، والآن لنقوم باستخدام هاذين المتغيرين، كالتالي:





كما نلاحظ استطاع كلا المتغيرين الوصول إلى البيانات التي قمنا بعمل وصف لهم في interface & class، وايضاً قمنا بمساعدة مترجم Typescript لفهم النوع الخاص بهذا المتغير.

كما نستطيع ان نعرف المتغير على انه مصفوفة من interface او class في حال كُنا نستقبل البيانات من الخادم على شكل مصفوفة، كالتالي:



نلاحظ اصبح المتغير user1 عبارة عن مصفوفة، ونستطيع تطبيق جميع مفاهيم المصفوفات عليها.

والخلاصة: نقوم بالاستفادة من interfaces وclassesg لتعريف أنواع جديدة لوصف البيانات القادمة لنا من الخادم بغرض افهام مترجم Typescript عن ما هي ونوع هذه البيانات والمتغيرات التي تُشير اليها، كما ايضاً تساعدنا بتجنب الأخطاء اثناء كتابة الشفرات البرمجية، وفي حالة كانت البيانات القادمة من الخادم على شكل مصفوفة نُعرف المتغير على انه مصفوفة من النوع interface او class، اما اذا كان وحيدة كان تكون بيانات مستخدم واحد فلا نحتاج إلى تعريفها كمصفوفة.

3- الدوال (العمليات) الأساسية في Angular HttpClient:

بداية وقبل التطرق إلى هذه العمليات يجدر الإشارة ان هذه الدوال مشابهة لدوال HTTP التي تم ذكرها في بداية هذا الكتاب، حيث قام مطورو Angular ببناء هذا المديول المسمى HttpClient وأضافوا اليه مجموعة من الدوال لها نفس أسماء دوال HTTP مثل (get-put-delete-..etc.) وتقوم ايضاً بنفس المهام، وايضاً يجب الإشارة أننا مطورو Front-End وهذا معناه اننا لا نختار هذه العمليات بشكل عشوائي في تطبيقنا وانما يقوم مطورو Back-End ببناء الAPIs ومن ثم يقومون بتحديد كل API والدالة المناسبة له، ونقوم نحن بأخذ هذه الAPIs مع الدوال الخاصة بها وتنفيذها كما هي، مثلاً رابط API معين نستخدم معه الدالة get بينما رابط آخر نستخدم معه الدالة put ورابط آخر نستخدم معه الدالة patch وهكذا باقي الدوال وقد يتكرر استخدام دالة معينة مع أكثر من روابط API مختلفة، وجميع هذه الAPIs ودوالها تأتينا مكتوبة وموضحة من مطوري Back-End وكل الذي نقوم به نحن هو استخدامها مع كل طلب Request لجهاز الخادم Server.

اما الآن لنقوم بإنشاء مشروع Angular جديد وليكن اسمه HttpClient-Demo والخطوة الثانية وللاستفادة من مميزات الHttp Client هي إضافة Module الخاص بها، والمُسمى HttpClientModule إلى App Module الرئيسي للمشروع في

حالة كان Logic المبني على هذا Module في Service (وهو المشهور والأكثر استخداماً كما سوف نشاهد لاحقاً) اما في حالة كنت عزيزي المتعلم تقوم بإضافة Logic في الComponent مباشرة وبنفس الوقت تستخدم ميزات Lazy Loading ففي هذه الحالة لابد من إضافة هذا Module في الModule الخاص بهذه الComponent، ولفهم اعمق لكيفية التعامل مع Modules الرجاء الاطلاع على كتاب Angular Routing And Modules، اما في حال الرغبة بالاستزادة بميزات Services الرجاء الاطلاع على كتاب Angular Component And Services، وكلا هاذين الكتابين ينتميان إلى هذه السلسلة.

اما الآن لنقوم بإضافة الHttpClientModule في App Module، كالتالي:

ملف app.module.ts

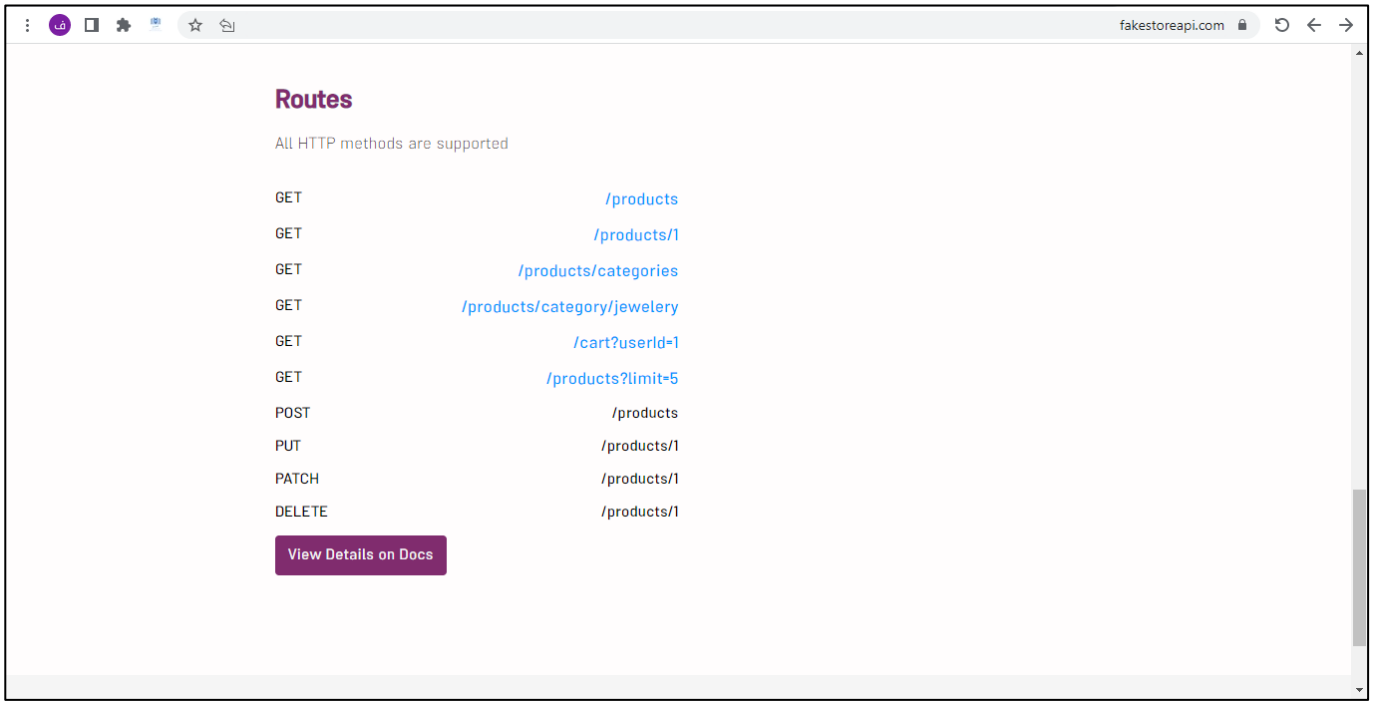
```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

وبذلك نكون اضفنا هذا Module والتي عن طريقها نستطيع الاستفادة من الكم الهائل من المميزات التي يقدمها وأولها العمليات الأساسية الأربعة في أي اتصال Http وهي (get – post – put - delete) والتي تُعيد لنا Observable لذلك نستطيع الاستفادة من جميع ميزات مكتبة Rxjs، اما الآن لنقوم باستعراض هذه الدوال كالتالي: (ملاحظة: بما اننا هنا ليس لدينا Back-End لذلك مبدئياً سوف نستخدم دوال API مجانية من موقع https://fakestoreapi.com ولاحقاً إذا احتجنا دوال متقدمة ومعقدة قد نستخدم مواقع أخرى)

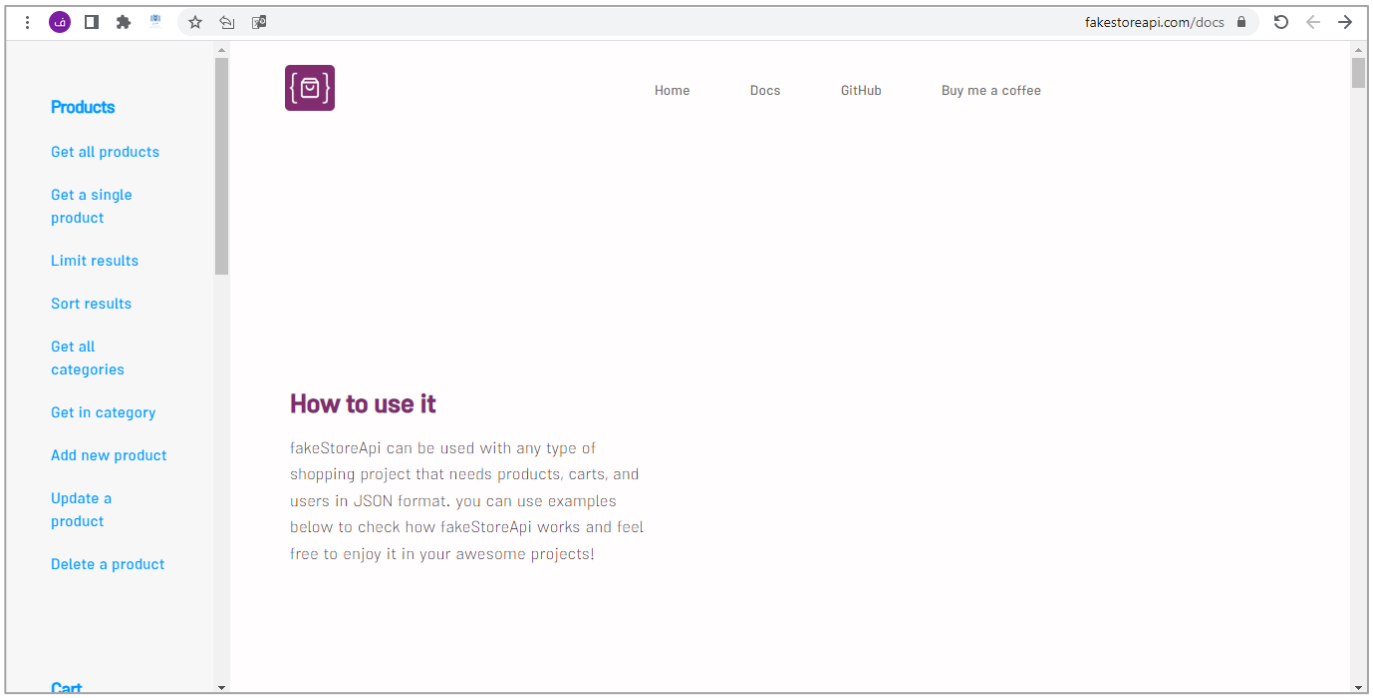
3-1- الدالة GET:

حيث تقوم هذه الدالة بجلب البيانات من الخادم Server بصيغة JSON بشكل افتراضي مع إمكانية تعديل الصيغة في حال الرغبة بذلك سواء نص او كائن او غيره وهو ما سوف نتطرق له لاحقاً بإذن الله، ولكن ما يهمنا هنا بالتحديد هو كيفية الاستخدام الأساسي لهذه الدالة لجلب البيانات.

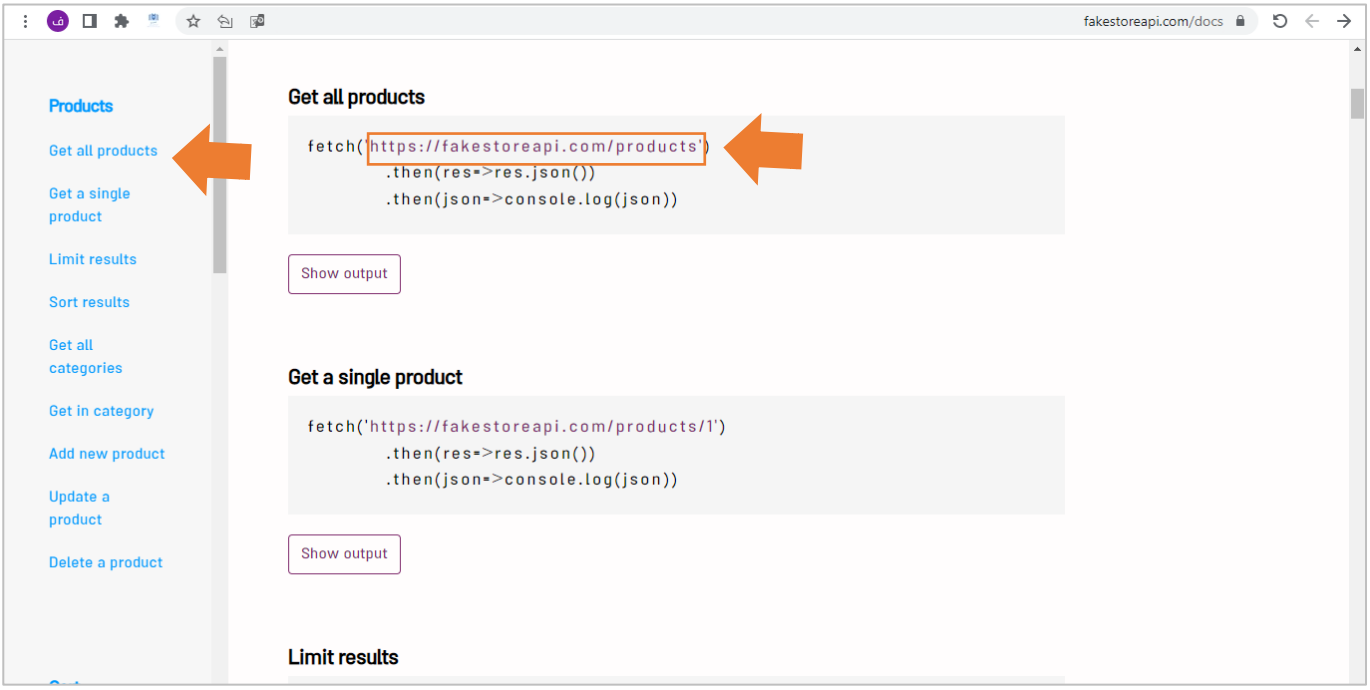
لذلك في البداية لنفتح موقع https://fakestoreapi.com ونستعرض الAPIs الموجودة فيه، كالتالي:



نلاحظ في الصورة السابقة عند النزول لأسفل الصفحة سوف نشاهد مجموعة من الدوال المدعومة مع روابط API الخاصة بها، لذلك لنضغط على زر View Details on Docs، لتظهر لنا جميع الدوال، كالتالي:



ومن الصفحة السابقة لنختار مثلاً Get All Products، وهي عبارة عن API نستخدم معها دالة GET لنستعرض جميع بيانات المنتجات Products (بيانات غير حقيقة وانما وضعت للاختبار والتجربة فقط)، كالتالي:



نلاحظ الرابط وهو عبارة عن API المستهدف الذي نطلبه من السيرفر لجلب البيانات، مع ملاحظة انه في حال لم يتم تحديد الدالة فإنها بشكل افتراضي ستكون GET، لذلك لنجلب هذا الرابط ونذهب إلى مشروعنا الذي قمنا بإنشائه سابقاً وبالتحديد في الComponent ذو الاسم app.component.ts ونضيف الشفرة البرمجية التالية:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {

  constructor(private http: HttpClient) {}

  ngOnInit() {}
}
```

نقوم بعمل Inject في متغير اسميته http، حيث عن طريقه نستطيع الوصول إلى جميع الدوال

اما ألان لنستخدم هذا المتغير للوصول إلى الدالة GET ونمرر لها الرابط الذي اخذناه من الموقع السابق كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
```

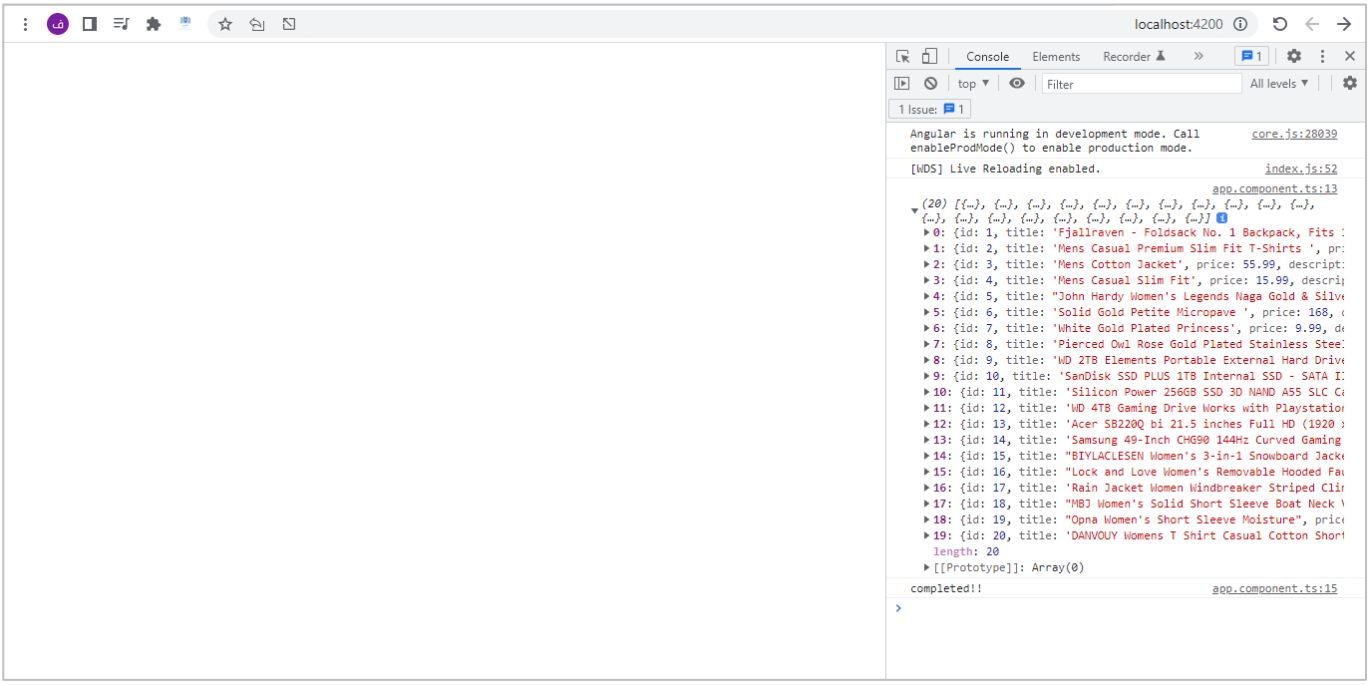
```
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.get('https://fakestoreapi.com/products');
  }
}
```

ولو قمنا بعمل serve للمشروع لمشاهدة النتيجة فسنجد انه لن يظهر لنا شيء، والسبب في ذلك عزيزي المتعلم ان الدالة get في HttpClient تقوم بإرجاع Observable وكما تعلمنا سابقاً انه لابد ان نعمل Subscribe لكي نشاهد محتويات هذا الObservable، وهذا الObservable هو الResponse او الرد القادم الخادم (السيرفر - server) وهو يختلف بحسب الدالة المستخدمة وبحسب ما قاموا به مطوري Back-End، وفي حالتنا هذه (GET) فإن الرد سيكون البيانات المُراد جلبها من السيرفر، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.get('https://fakestoreapi.com/products').subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed!!'),
    });
  }
}
```

والآن لنذهب إلى المتصفح ولنشاهد النتيجة، كالتالي:



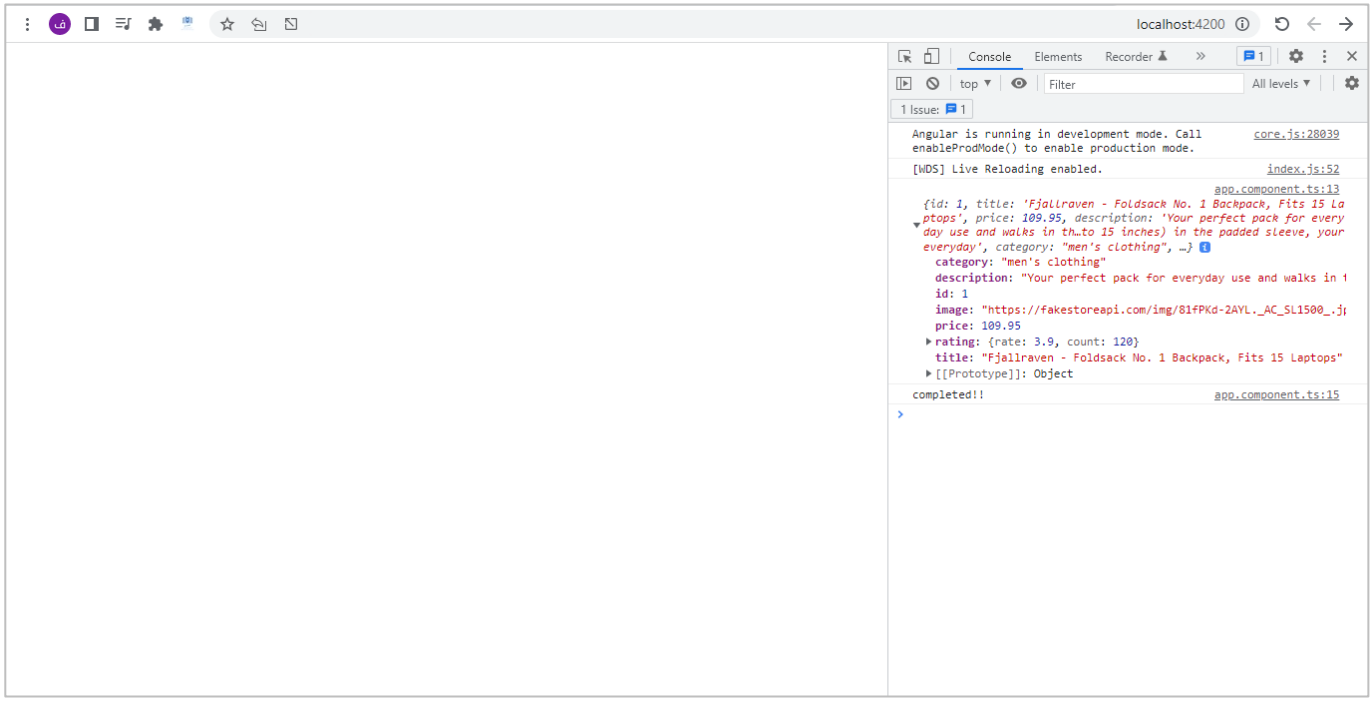
نلاحظ تم جلب البيانات عن طريق رابط api باستخدام الدالة get، وبما ان هذه الدالة تُعيد Observable فنستطيع استخدام معها جميع دوال مكتبة RXJS التي تعلمناها سابقاً.

كما ايضاً نستطيع جلب بيانات محددة كان تكون بيانات product معين، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.get('https://fakestoreapi.com/products/1').subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed!!'),
    });
  }
}
```

نلاحظ قمنا بتمرير قيمة في الرابط بشكل ثابت وهو رقم (1) وهذا الرقم هو عبارة عن id لproduct معين (هذا id لم نمرره اعتباطاً وانما بحسب ما يأتينا من مطوري Back-End بحيث يتم استخدام API معين يستقبل بارامتر للوصول إلى بيانات محددة)، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



كما نستطيع تمرير هذا البارامتر للAPI بشكل ديناميكي، كالتالي:

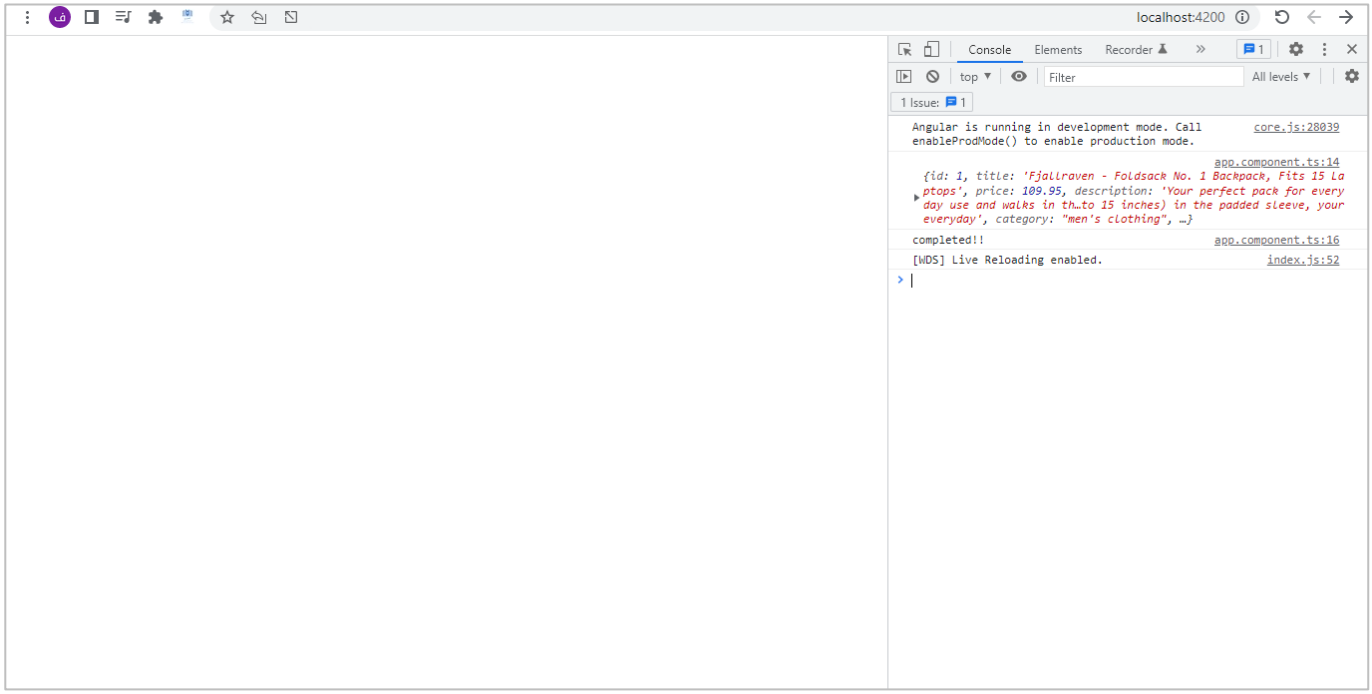
ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    const id = '1';
    this.http.get(`https://fakestoreapi.com/products/${id}`).subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed!!'),
    });
  }
}
```

كما نلاحظ عرفنا متغير باسم id ومررناه إلى رابط API بشكل ديناميكي عن طريق استخدام ميزة String Interpolation الخاصة بجافا سكريبت، وتستطيع استخدام الطريقة التقليدية في حال اردت ذلك.

```
this.http.get('https://fakestoreapi.com/products/' + id).subscribe({
```

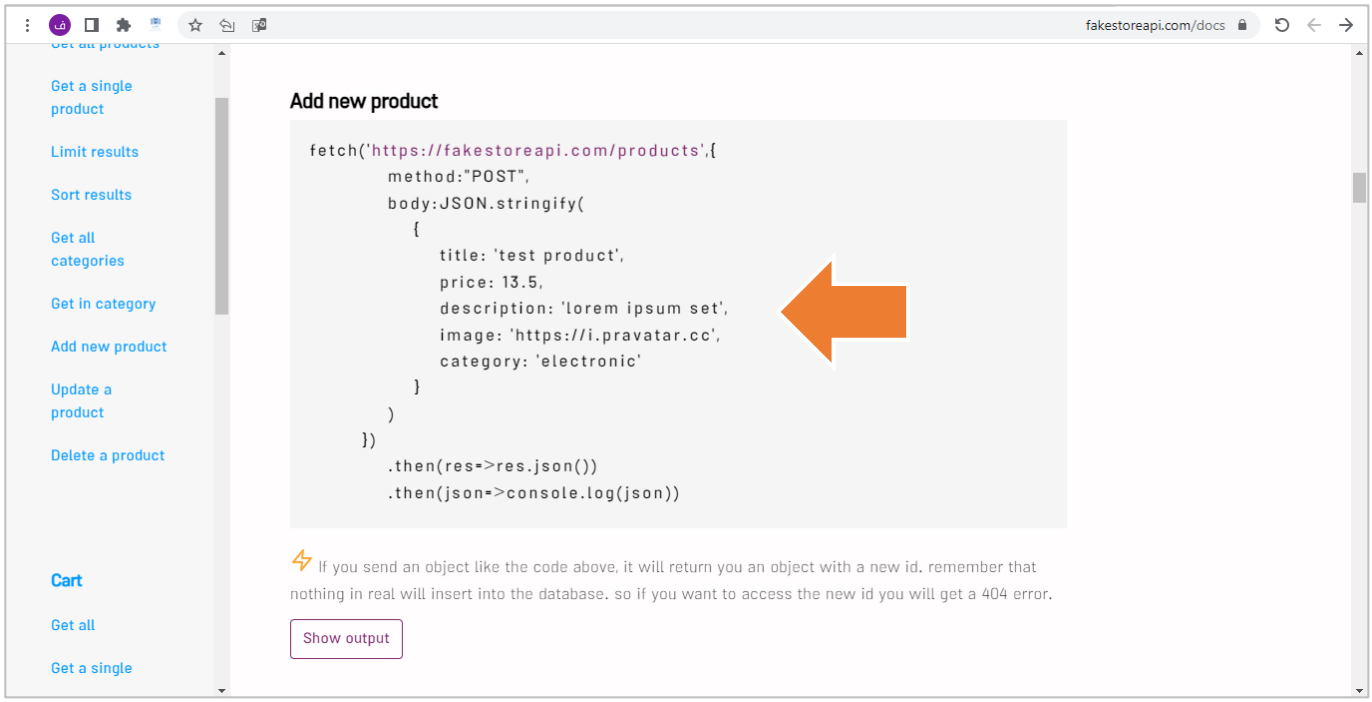
والنتيجة ستكون كالتالي:



وبذلك تعرفنا كيفية جلب البيانات والاستخدامات الأساسية لهذه الدالة، وفي الجزء التالي سوف نتعلم كيفية التعامل مع الدالة POST.

3-2- الدالة POST:

نستخدم هذه الدالة لإضافة البيانات، ونمرر لها رابط API، ونمرر لها ايضاً البيانات التي نُريد إضافتها (body) على شكل كائن JavaScript، والآن لنذهب إلى موقع Fake API Store وبالتحديد إلى جزء POST، كالتالي:

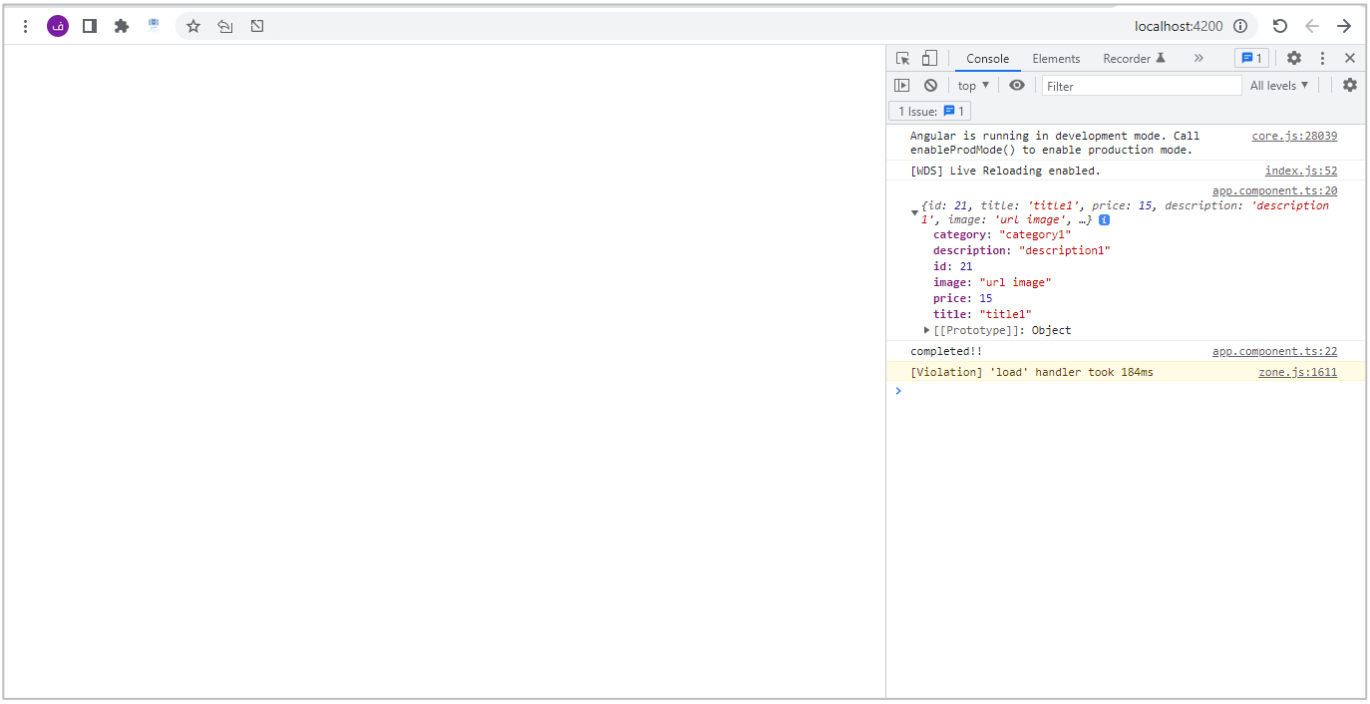


نلاحظ body المراد اضافته هو title تستقبل قيمة نصية وprice تستقبل قيمة رقمية description, تستقبل قيمة نصية وimage تستقبل قيمة نصية وcategory تستقبل قيمة نصية، وهذا body لا نكتبه اعتباطاً وانما يتم تجهيزه لنا من قبل مطوري Back-End والذي نقوم به فقط هو تمرير القيم وارسالها على شكل طلب Request إلى السيرفر، كالتالي:

ملف app.component.ts

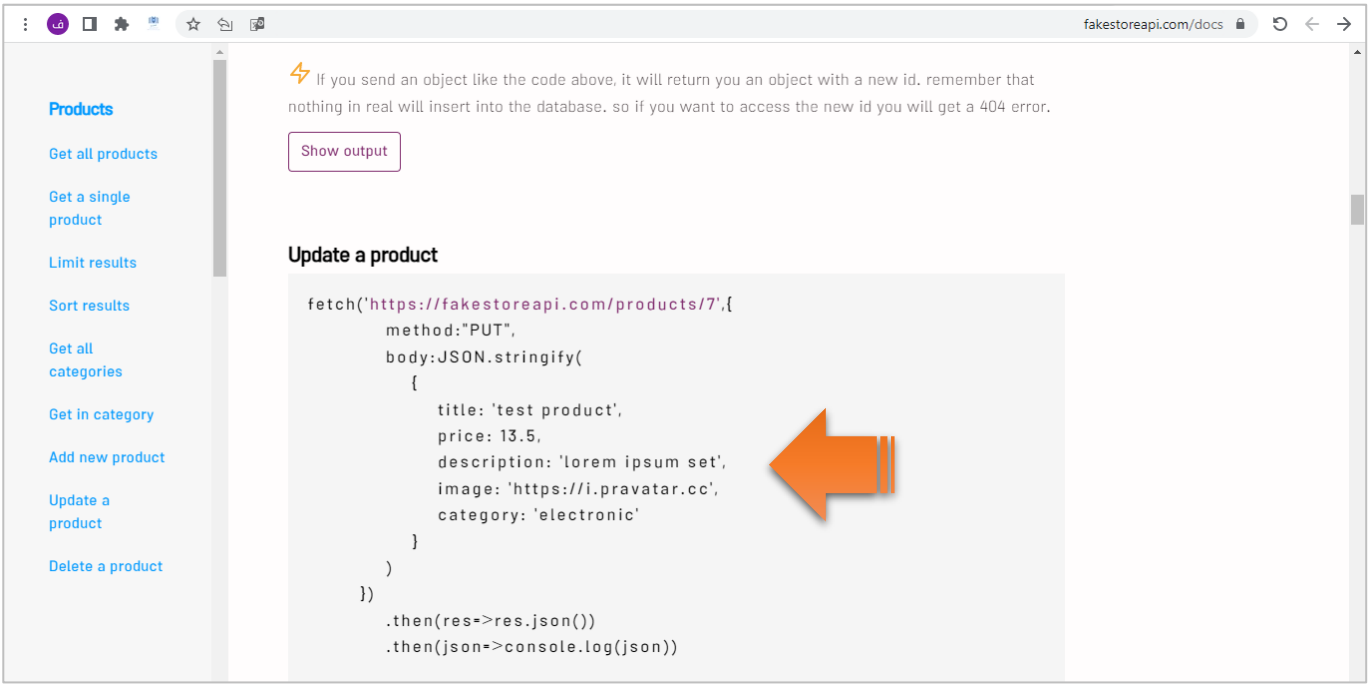
```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .post('https://fakestoreapi.com/products', {
        title: 'title1',
        price: 15,
        description: 'description1',
        image: 'url image',
        category: 'category1',
      })
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

اما النتيجة فتكون كالتالي:



3-3- الدالة PUT:

اما هذه الدالة فنستخدمها لتعديل البيانات (UPDATE)، وهي نفس الدالة POST تستقبل البارامتر الأول وهو عبارة عن رابط API اما الثاني فهو body او البيانات المراد التعديل عليها، اما الآن لنذهب إلى نفس الموقع السابق وبالتحديد لجزء تعديل product، كالتالي:



ونفس ما قيل في POST يُقال هنا ولعل الفرق اننا هنا نقوم بالتعديل على بيانات موجودة بعكس POST التي عن طريقها نُضيف بيانات جديدة، والآن لنقوم بتطبيق الشفرة البرمجية، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
```

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .put('https://fakestoreapi.com/products/7', {
        title: 'title2',
        price: 20,
        description: 'description2',
        image: 'url image',
        category: 'category1',
      })
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

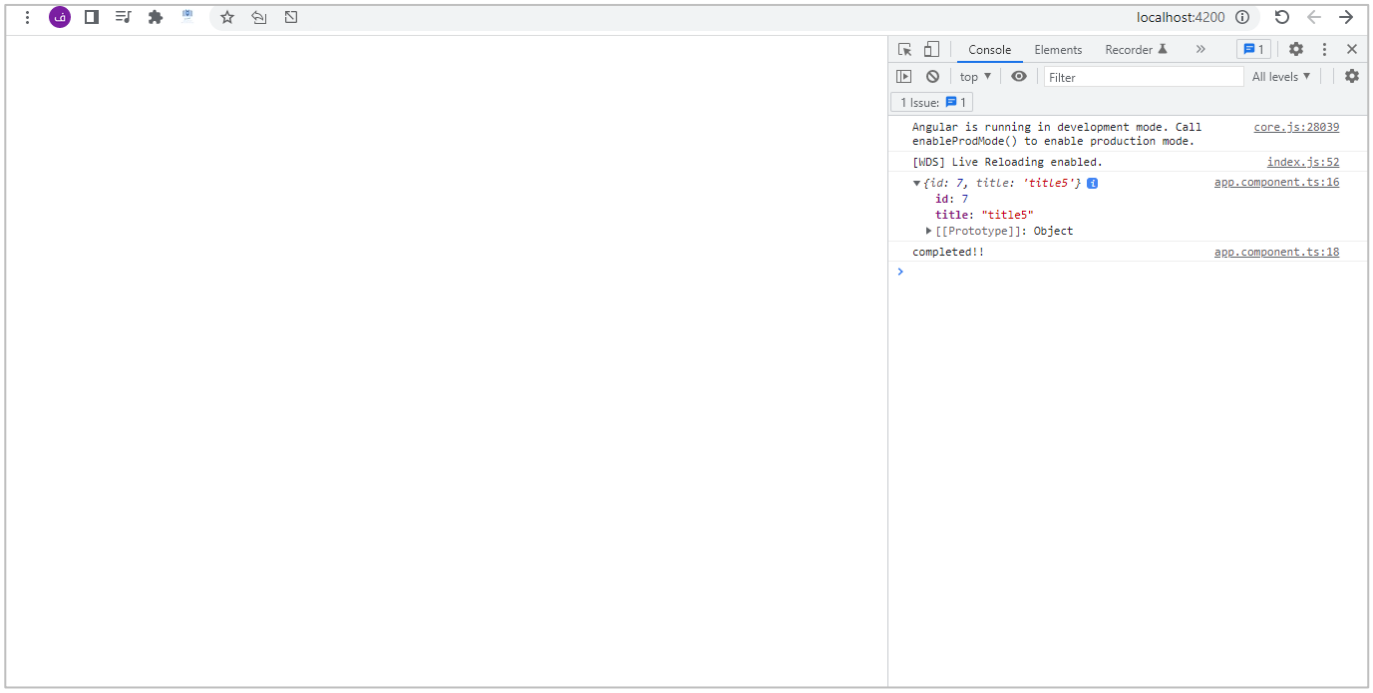
4-3- الدالة PATCH:

وهذه الدالة مشابهه لدالة PUT والفرق انه يتم استعمالها لتعديل جزء معين من البيانات فقط، مع العلم انه يمكن استخدام الدالة PUT لتعديل جزء معين من البيانات، وهذا الأمر يرجع بالنهاية إلى مطوري Back-End وما يتم ارساله لنا كمطوري Front-End من تعليمات.

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .patch('https://fakestoreapi.com/products/7', {
        title: 'title5',
      })
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
```

```
        complete: () => console.log('completed!!'),
      });
  }
}
```

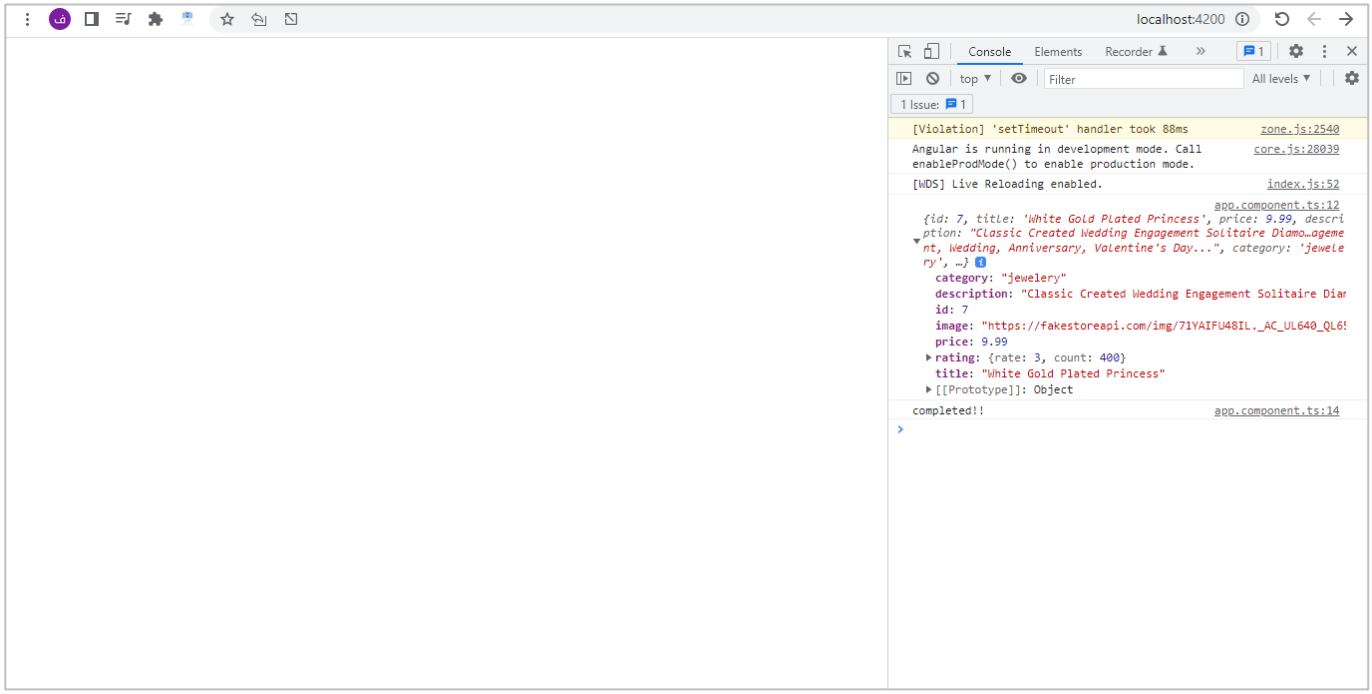


5-3- الدالة DELETE:

ومن اسمها نستخدمها لحذف البيانات، ونمرر لها فقط رابط API للبيانات المراد حذفها، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.delete('https://fakestoreapi.com/products/7').subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed!!'),
    });
  }
}
```



كما نلاحظ ان Response (الاستجابة - الرد) الذي اتى من الخادم (السيرفر) هو الكائن المراد حذفه، وهذا الرد يختلف ففي اغلب الأحيان لا يرد السيرفر باي بيانات فقط بالرمز 200 أي تمت العملية بنجاح.

4- Explicitly Type Response/Request :

جميع لدوال السابقة هي عبارة عن generic type أي نستطيع التصريح عن نوع البيانات في هذه الدوال سواء عن طريق Response او Request، لذلك نستطيع تعريف هذه البيانات عن طريق الInterfaces/classes كما ذكرنا سابقاً في جزء interfaces/classes حيث نستطيع التصريح او تعريف البيانات القادمة عن لنا عن طريق Response او الخارجة منا عن طريق Request، وهذا التعريف يتم بناءً على ما يقدمه لنا مطوري Back-End، من تعليمات، فمثلاً في حالة جلب بيانات الproducts بواسطة الدالة GET فنحن نتوقع - بناءً على ما تعاملنا معه من خلال موقع Fake API Store – ان تكون البيانات بالشكل التالي: (id – title – description – image – price - category) بالإضافة إلى rating ويحتوي على اثنين خاصية وهما rate وcount، لذلك نستطيع تعريف هذه البيانات عن طريق انشاء interface، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

export interface Product {
  id: number;
  title: string;
  description: string;
  image: string;
  price: number;
  category: number;
  rating: {
    rate: number;
    count: number;
```

```
  };
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.get('https://fakestoreapi.com/products/').subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed!!'),
    });
  }
}
```

طبعاً يُفضل انشاء interface في ملف منفصل ومن ثم استدعائه في أي component نُريده.

كما نلاحظ انشئنا interface وعرفنا فيه مجموعة من الخصائص تحمل نفس اسم خصائص البيانات المتوقع رجوعها لنا من الخادم (Response)، ولكي يكون عملنا اكثر احترافية نستطيع فصل خاصية rating في interface لوحده وليكن اسمه ProductRating، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

export interface Product {
  id: number;
  title: string;
  description: string;
  image: string;
  price: number;
  category: number;
  rating: ProductRating;
}

export interface ProductRating {
  rate: number;
  count: number;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
```

```
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.get('https://fakestoreapi.com/products/').subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed!!'),
    });
  }
}
```

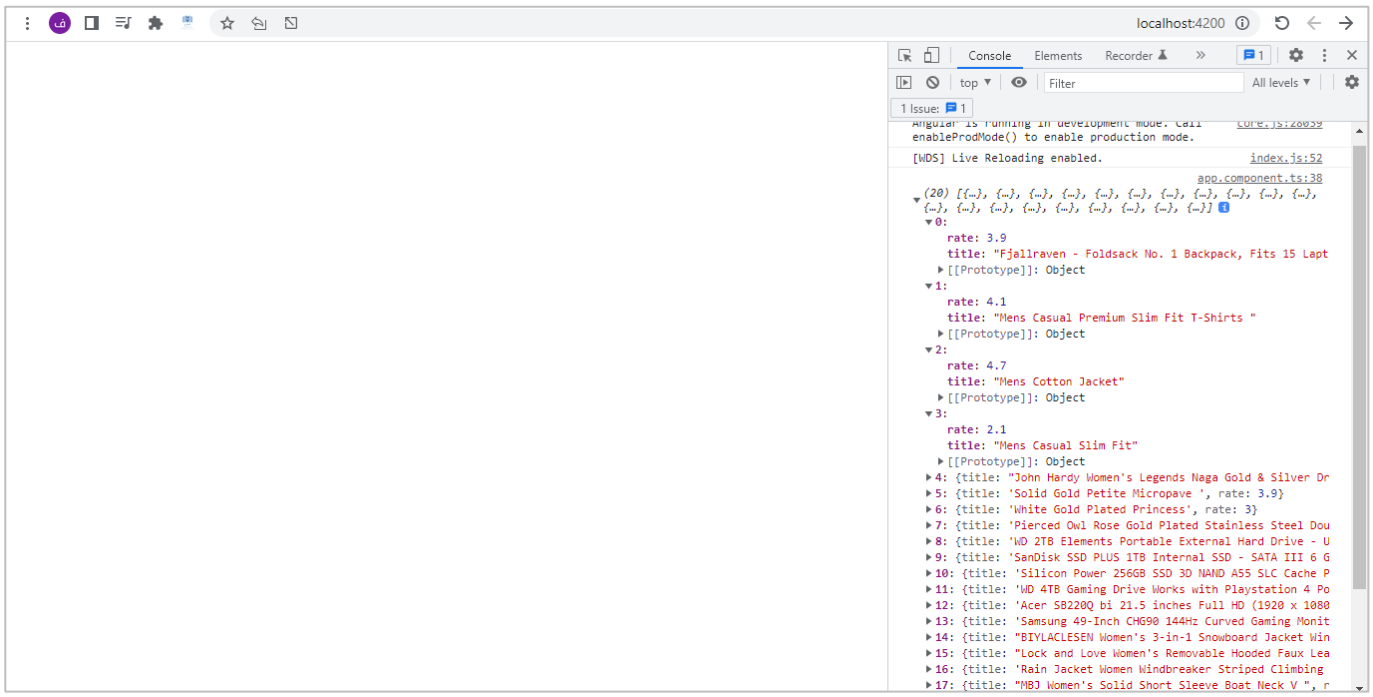
والآن نستطيع استخدام هذا Interface، وليكن مثلاً نُريد استخراج اسم المنتج والتقييم rate الخاص به فقط من البيانات القادمة لنا، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { map } from 'rxjs/operators';

export interface Product {
  id: number;
  title: string;
  description: string;
  image: string;
  price: number;
  category: number;
  rating: ProductRating;
}

export interface ProductRating {
  rate: number;
  count: number;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get<Product[]>('https://fakestoreapi.com/products/')
      .pipe(
        map((data) =>
          data.map((val) => {
            return { title: val.title, rate: val.rating.rate };
          })
```

```
        )
      )
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

والنتيجة:



Observe Response -5:

بشكل افتراضي Response القادم لنا من السيرفر من خلال اتصال HTTP يكون الBody – أي البيانات التي يحملها – ولكن في بعض الأحيان نُريد ان نقرأ Response كاملاً مشتمل على Header والBody بالإضافة إلى بعض الخصائص الأخرى.

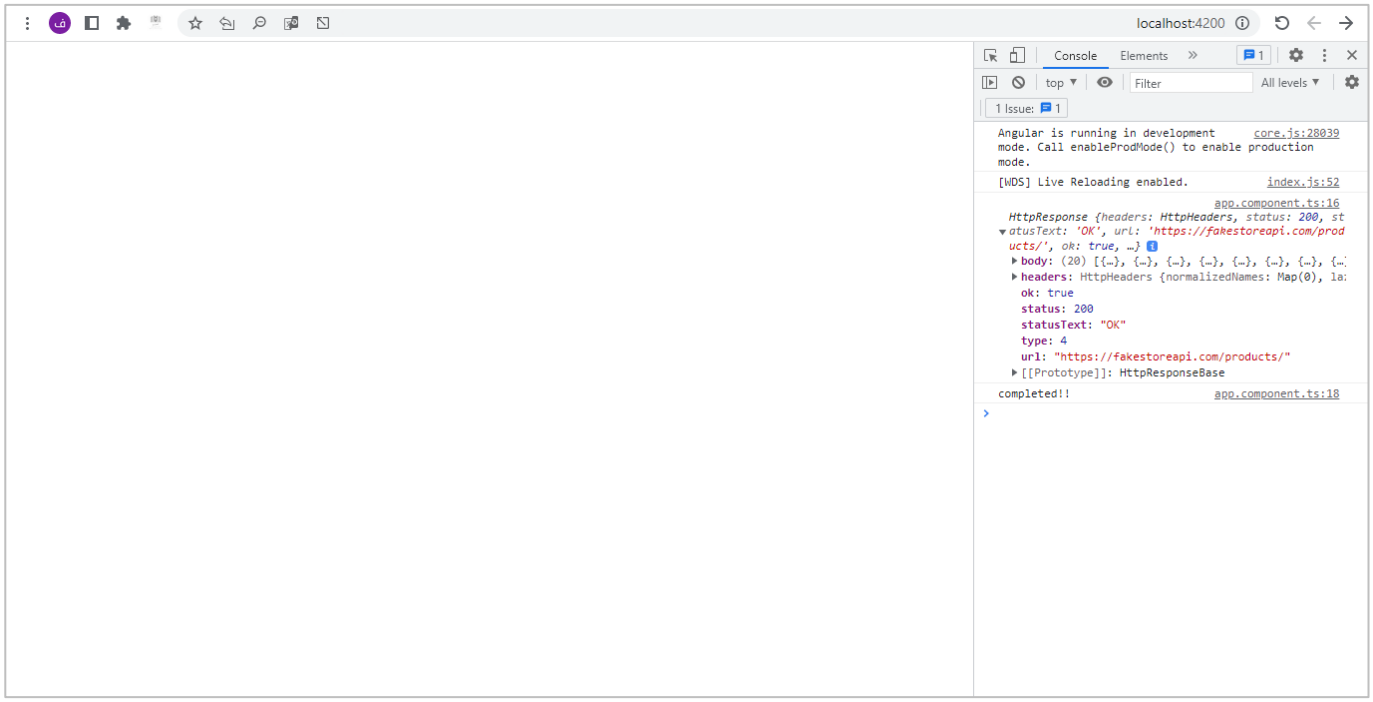
وللقيام بهذا الأمر نقوم بتغيير قيمة الخاصية observe إلى response، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
```

```
  ngOnInit() {
    this.http
      .get('https://fakestoreapi.com/products/', {
        observe: 'response',
      })
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

والنتيجة في المتصفح تكون كالتالي:



كما نلاحظ الResponse هو عبارة عن Body والHeader وهو من النوع HttpHeaders – نوع بيانات خاص بالAngular يصف الخصائص التي يحملها الHeader بالإضافة إلى احتوائه على مجموعة من الدوال والخصائص لتحكم بهذا الHeader – بالإضافة إلى بعض الخصائص التي تصف هذا الاتصال مثل حالة الاتصال (status) او حالة الطلب (ok) او رابط api (url).

6- Observe Events:

من الميزات التي يقدمها لنا Angular HttpClient هي إمكانية قراء جميع احداث الاتصال بالخادم Server (مراحل الاتصال) بدأً من ارسال Request ومن ثم استقبال الHeader الخاص بالResponse ومن ثم الBody للResponse وايضاً هنالك أحداث ومراحل رفع الملف على الخادم server او تحميل الملف من الخادم server ولعل آخر ميزتين(تحميل/رفع ملف) هما المستخدمتين بكثرة، بحيث نستخدمها لعرض Progress للمستخدم تُبين له تقدم تحميل او رفع الملف.

وجميع هذه الإحداث او مراحل الاتصال تم تعريفها في الAngular من خلا enum تم تسميته HttpEventType، ولو استعرضناه لوجدنا محتواه بهذا الشكل:

```
export declare enum HttpEventType {
    /**
     * The request was sent out over the wire.
     */
    Sent = 0,
    /**
     * An upload progress event was received.
     */
    UploadProgress = 1,
    /**
     * The response status code and headers were received.
     */
    ResponseHeader = 2,
    /**
     * A download progress event was received.
     */
    DownloadProgress = 3,
    /**
     * The full response including the body was received.
     */
    Response = 4,
    /**
     * A custom event from an interceptor or a backend.
     */
    User = 5
}
```

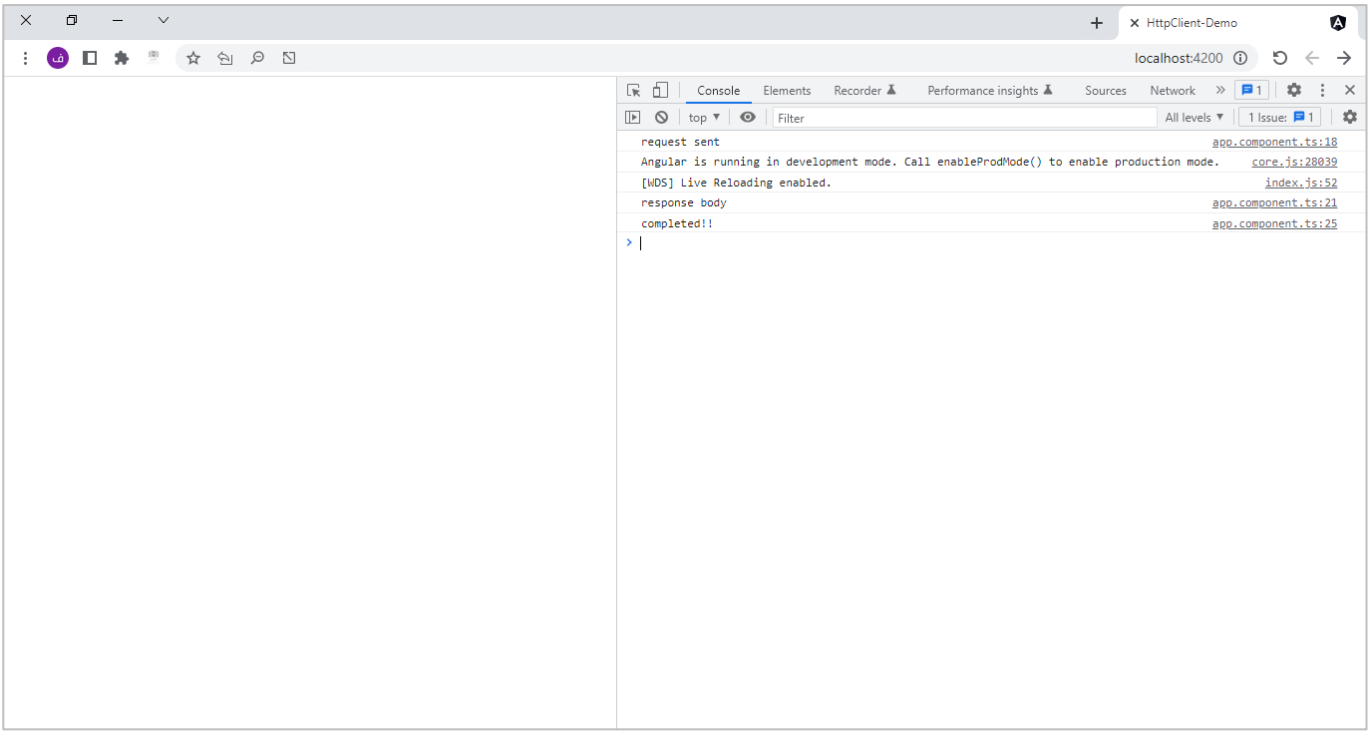
حيث sent ترمز إلى ارسال الاتصال إلى Server وهو ما نسميه Request وUplaodProgress نستخدمه لمعرفة تقدم رفع الملف إلى السيرفر في حال كان هنالك ملف يتم رفعه، وResponseHeader نستخدمه لمعرفة انه تم استقبال الHeader من الخادم – السيرفير – وDownloadProgress لمعرفة مدى تقدم تحميل الملف من الخادم، وResponse ونقصد بها انه تم استلام Body – وفي الغالب تكون بيانات على شكل JSON – من الخادم.

ولتوضيح لنعطي المثال التالي، حيث لابد في البداية ان نقوم بتغيير الObserve إلى events، كالتالي:

app.component.ts ملف

```
import { HttpClient, HttpEventType } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get('https://api.github.com/users', {
        observe: 'events',
      })
      .subscribe({
```

```
        next: (event) => {
          if (event.type === HttpEventType.Sent) {
            console.log('request sent');
          }
          if (event.type === HttpEventType.Response) {
            console.log('response body');
          }
        },
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```



اما الآن لنتطرق إلى الجزئية المهمة وفي الحقيقة هي ما يهمنا في ميزة Observe Events وهي معرفة تقدم ملف سواء كان يتم رفعه إلى الخادم او تنزيله من الخادم.

1-6- Listening for Progress:

سوف نتكلم في هذه الجزئية عن كيفية الاستفادة من ميزة Observe Events لمراقبة والاستماع لمدى تقدم رفع ملف إلى الخادم وإظهار Progress Bar للمستخدم لكي نبين له مدى تقدم رقع الملف.

اول شيء لابد من تفعيل خاصية أخرى وهي reportProgress ونجعل قيمته true، ثانياً لنعطي المثال ومن ثم نشرح الجزئيات التي تحتاج إلى شرح وتعليق، كالتالي:

ملف app.component.ts

```
<div>
  <input type="file" (change)="onChange($event)" style="display: block" />
```

```
  <button (click)="onUpload()" style="margin-top: 40px">Upload</button>
</div>
<div style="margin-top: 10px" *ngIf="progress">
  <progress [value]="progress" max="100" style="height: 20px"></progress>
  <span style="padding-left: 10px">{{ progress }} %</span>
</div>
```

في هذا الملف وضعنا Markup اللازم لإضافة ملف عن طريق الدالة onChange ورفع هذا الملف عن طريق الدالة onUpload، بطبيعة الحال لم اهتم بالتصميم لكي لا نتشتت عن ما نريد شرحه بالإضافة إلى انه خارج نطاق هذا الكتاب.

وما يهمنا هو المتغير progress الذي نخزن فيه مدى تقدم الملف وربطنا بإداة progress ولتحسين أكثر قمنا بطباعة هذه القيمة على شكل نسبة مئوية في التاغ span.

ملف app.component.ts

```
import { HttpClient, HttpEventType } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}

  progress = 0;
  file: any;

  ngOnInit() {}

  onChange(event: any) {
    this.file = event.target.files[0];
  }

  onUpload() {
    const formData = new FormData();
    formData.append('file', this.file, this.file.name);
    this.http
      .post('https://file.io', formData, {
        observe: 'events',
        reportProgress: true,
      })
      .subscribe((event) => {
        if (event.type === HttpEventType.UploadProgress) {
          if (event.total !== undefined)
            this.progress = Math.round((100 * event.loaded) / event.total);
        }
      });
  }
}
```

1 2 3 4 5

١. قمنا بتعريف متغيرين الأول لتخزين مدى تقدم الملف والثاني لتخزين الملف الذي اختاره المستخدم.

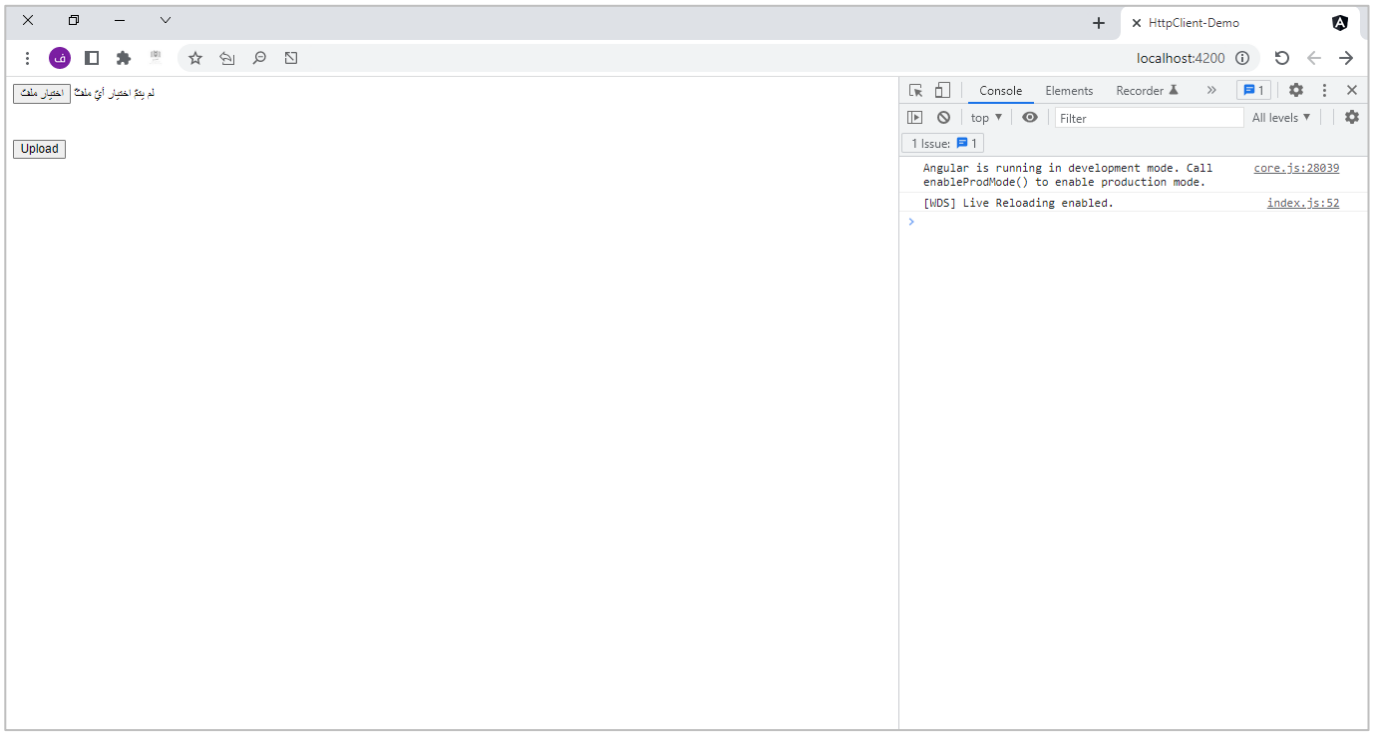
٢. نفذنا محتوى الدالة onChange وتقوم بقراءة الملف المختار وتخزينه في المتغير file الذي تم تعريفه في الخطوة السابقة.

٣. عرفنا متغير من النوع FormData لكي نستطيع ارساله إلى عن طريق API، ولفهم أكثر الرجاء مراجع كتابي Angular Forms.

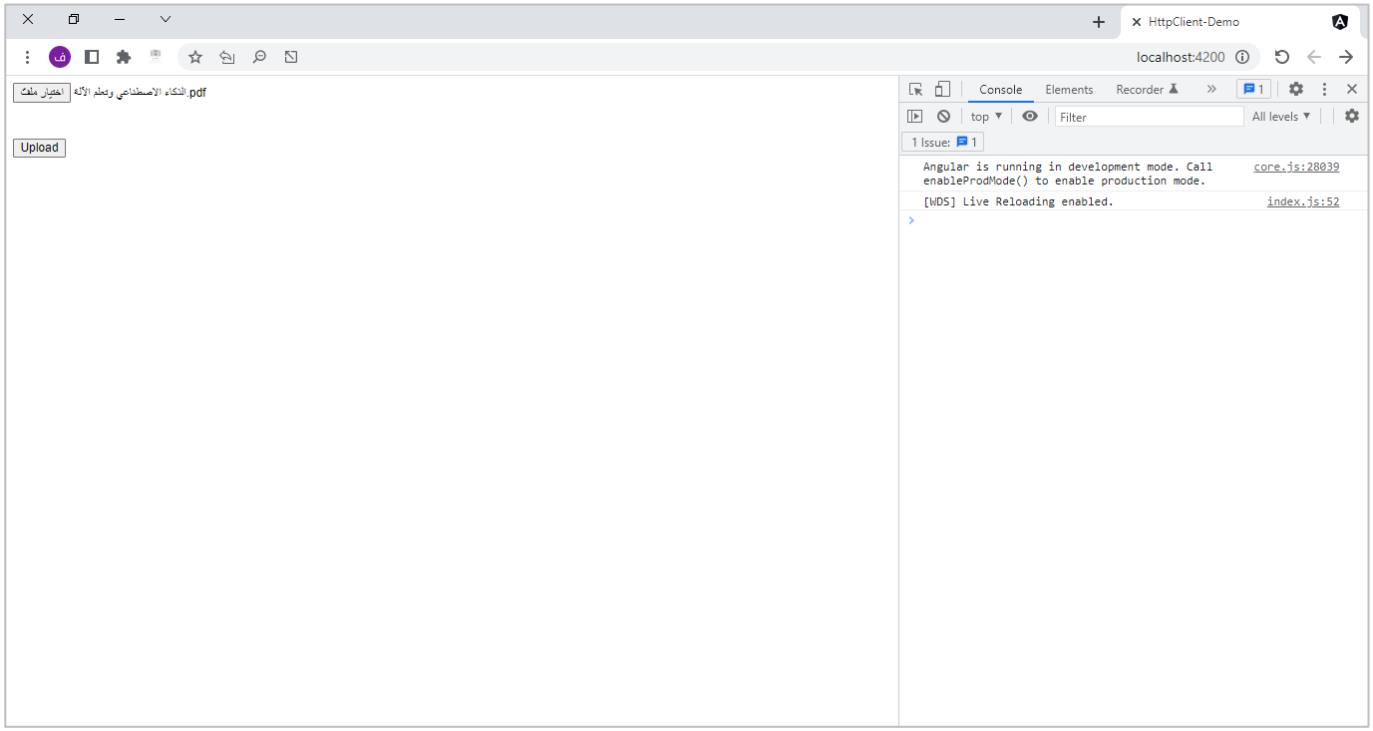
٤. اضفنا الخاصية reportProgress وجعنا قيمتها true، لكي نستطيع قراءة تقدم الملف في حال الرفع او تحميل الملف إلى الخادم.

٥. وهنا قمنا بوضع شرط لتأكد من ان حالة اتصال Http (Event) هو رفع الملف، فإذا كان كذلك نقوم بعمل عملية حسابية بسيطة لمعرفة مدى تقدم الملف ومن ثم تخزين القيمة في المتغير progress.

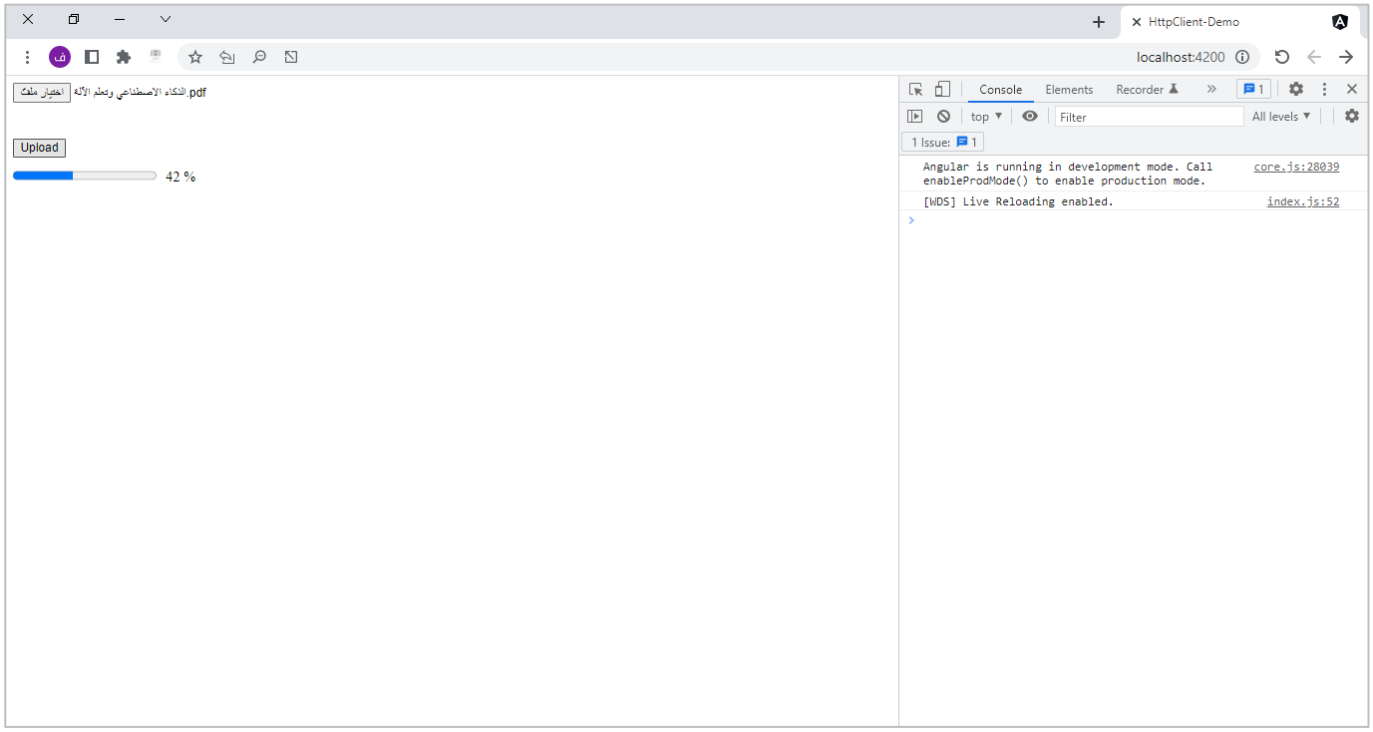
اما الآن لنشاهد النتيجة في المتصفح، كالتالي:

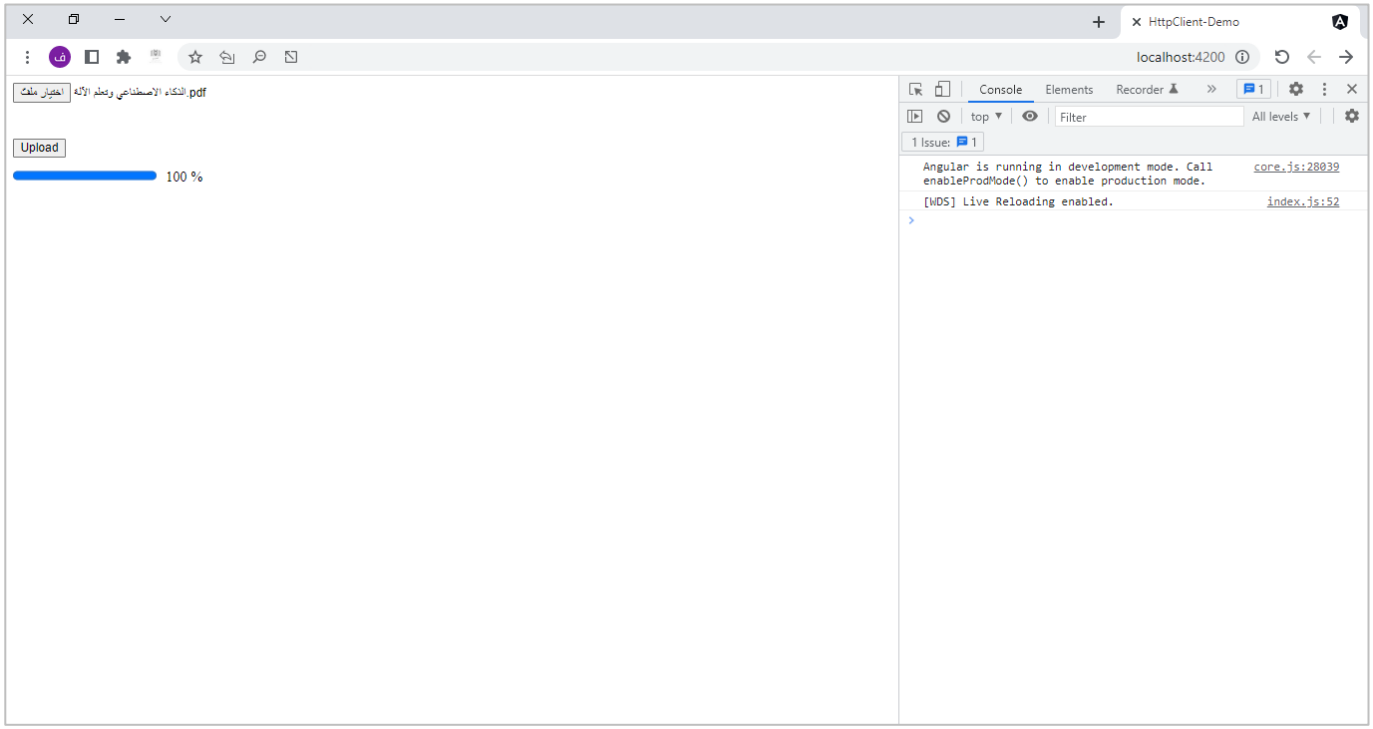


لنختار ملف وليكن حجمه متوسط لكي نستطيع ان نرى مدى التقدم بشكل واضح، كالتالي:



بعد اختيار الملف لنقوم بالضغط على زر upload لكي نرفع الملف على الخادم وبنفس الوقت نرى Progress Bar، كالتالي:





7- التحكم بأهم خصائص الHeader:

كما أشرنا سابقاً في بداية هذا الكتاب ان الHeader في اتصال HTTP يحتوي على مجموعة من الخصائص التي تصف هذا الاتصال مثل نوع البيانات او إضافة tokens والتعامل مع CROS وغيره من الخصائص، لذلك وفي هذا الجزء سوف نتطرق إلى اهم هذه الخصائص وكيفية التعديل عليها والتحكم بها، وايضاً يجب التنويه ان افضل طريقة لتعامل مع header في الAngular تتم من خلال ما يسمى Interceptors والتي سوف نتطرق لها لاحقاً، ولكن الآن سوف نتطرق على كيفية التحكم بهذه الخصائص بالشكل البسيط.

قدمت لنا Angular طريقتين لتحكم بهذه الخصائص، بحيث بعض الخصائص يتم اضافتها عن طريق الكلاس HttpHeaders وخصائص أخرى يتم اضافتها مباشرة على شكل كائن جافا سكريبت، وسوف نستعرض اهم هذه الخصائص وطرق اضافتها والتحكم بها، بالإضافة سوف نتطرق لمجموعة من الدوال الجاهزة التي قدمتها لنا Angular لتحكم أكثر بخصائص الHeader مثل has والتي تُعيد قيمة منطقية true او false ونستخدمها لاختبار هل توجد خاصية معينة في الHeaders، اما الآن لنتطرق لاهم خصائص الHeader وبعدها نستعرض هذه الدوال، كالتالي:

7-1- Response Type:

بشكل افتراضي الHttpClient تفترض ان نوع الاستجابة تكون البيانات فيها على هيئة JSON مع إمكانية تغيير نوع الاستجابة بحسب نوع البيانات التي نُريدها، ويمكن حصر الأنواع التي يدعمها HttpClient بالجدول التالي:

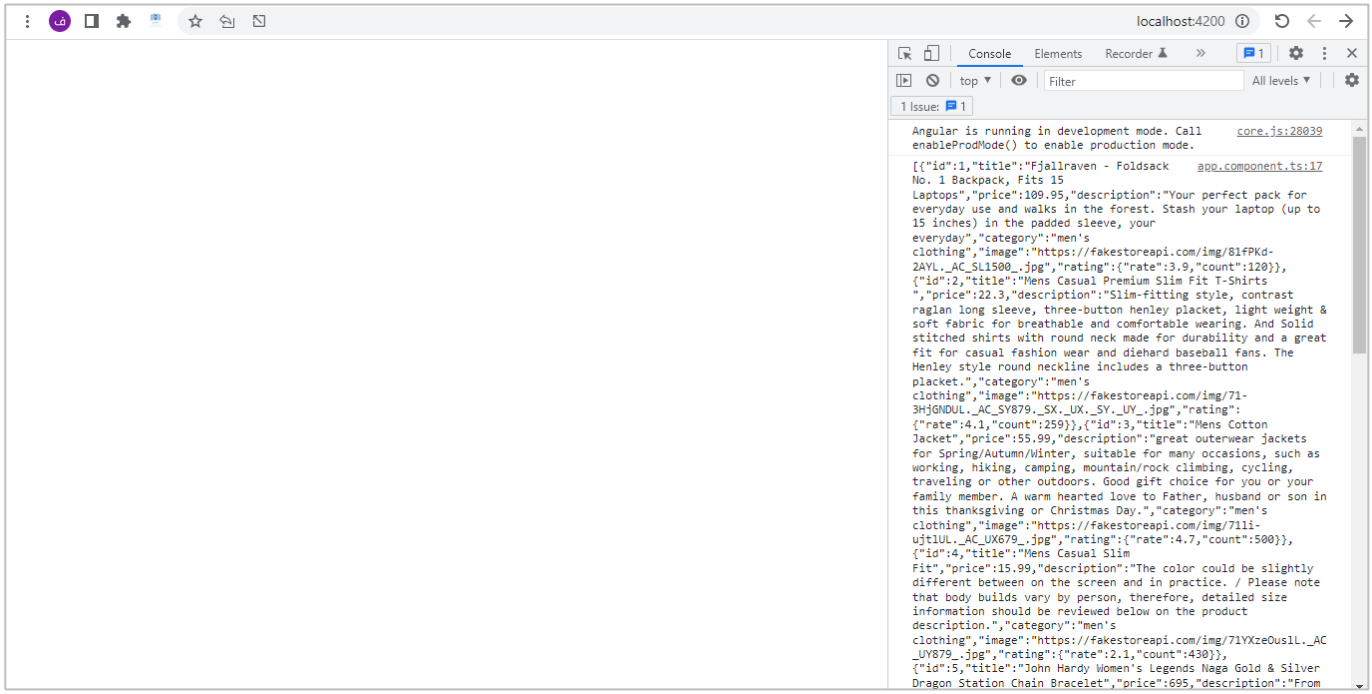
json	text	blob
بيانات على شكل صيغة JSON	بيانات نصية على شكل هيئة plain text	بيانات وسائط متعددة، صور فيديو، صوت،. الخ

ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get('https://fakestoreapi.com/products', {
        responseType: 'text',
      })
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

كما هو ملاحظ اضفنا بارامتر ثاني إلى – وهو اختياري – الدالة GET على شكل كائن جافا سكريبت حيث الخاصية هي responseType والقيمة text مع إمكانية تغيير هذه القيمة إلى blob في حال اردنا التعامل مع الوسائط او json في حال اردنا الصيغة تكون على شكل JSON مع العلم انا في حالة لم نضيف هذه الخاصية فإن Angular سوف يُعامل هذه البيانات على انها JSON بشكا افتراضي، اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



كما نلاحظ تم تغيير صيغة البيانات القادمة من JSON إلى Plain Text.

Add Tokens -2-7:

تكلمنا عن مفهوم Tokens في بداية هذا الكتاب، وهنا سوف نتطرق لطريقة إضافة هذا Token إلى أي طلب Request إلى الخادم Server، ويتم بكل بساطة من خلال الخاصية header والتي نمرر لها كائن وهذا الكائن يحتوي على أغلب خصائص الHeader حيث نمررها على شكل (key: value) بحيث يمثل الkey اسم الخاصية بداخل هذا الكائن والvalue قيمة هذه الخاصية، وما يهمنا من هذه الخصائص هي Authorization وهذه الخاصية نمرر لها قيمة token، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}

  ngOnInit() {
    const token = "ddagbhhhbhdhhgdbhtftuvjhvgftxchvjhvjhvfcxgtxf"
    this.http
      .get('https://fakestoreapi.com/products/', {
        observe: 'response',
        headers: {
          Authorization: `Bearer ${token}`,
        }
      })
      .subscribe({
```

```
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

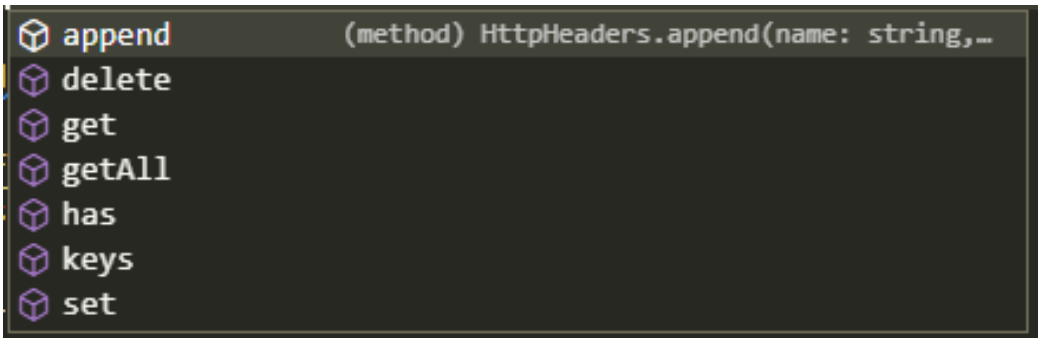
قمنا بمحاكاة ان لدينا token وقمنا بتخزينه في المتغير ذو الاسم token، ومن ثم قمنا بتمريره إلى header على شكل كائن وهذا الكائن هو الذي يحتوي على الخاصية Authorization وهي التي مررنا لها قيمة هذا token.

ملاحظة: من الخصائص الأخرى التي يحتويها هذا الكائن (content-type – Access-Control-Allow-Origin – ..الخ).

3-7- HttpHeaders Methods:

استكملاً لما بدأناه سابقاً في الخاصية header، حيث ان هذه الخاصية تم تعريفها على انها من النوع HttpHeaders وهو عبارة عن class يحتوي على مجموعة من الدوال التي نستطيع عن طريقها التحكم بخصائص Header في Request قبل أرساله إلى Server او مع Response في حال قدومه من Server مع العلم ان ليس كل خاصية نستطيع التعديل عليها وليس كل خاصية نستطيع اضافتها، فهذا يرجع إلى طريقة بناء الHeader في الBack-End.

وهنالك مجموعة من الدوال نستطيع استخدامها لتعامل مع الHeaders، وهي:



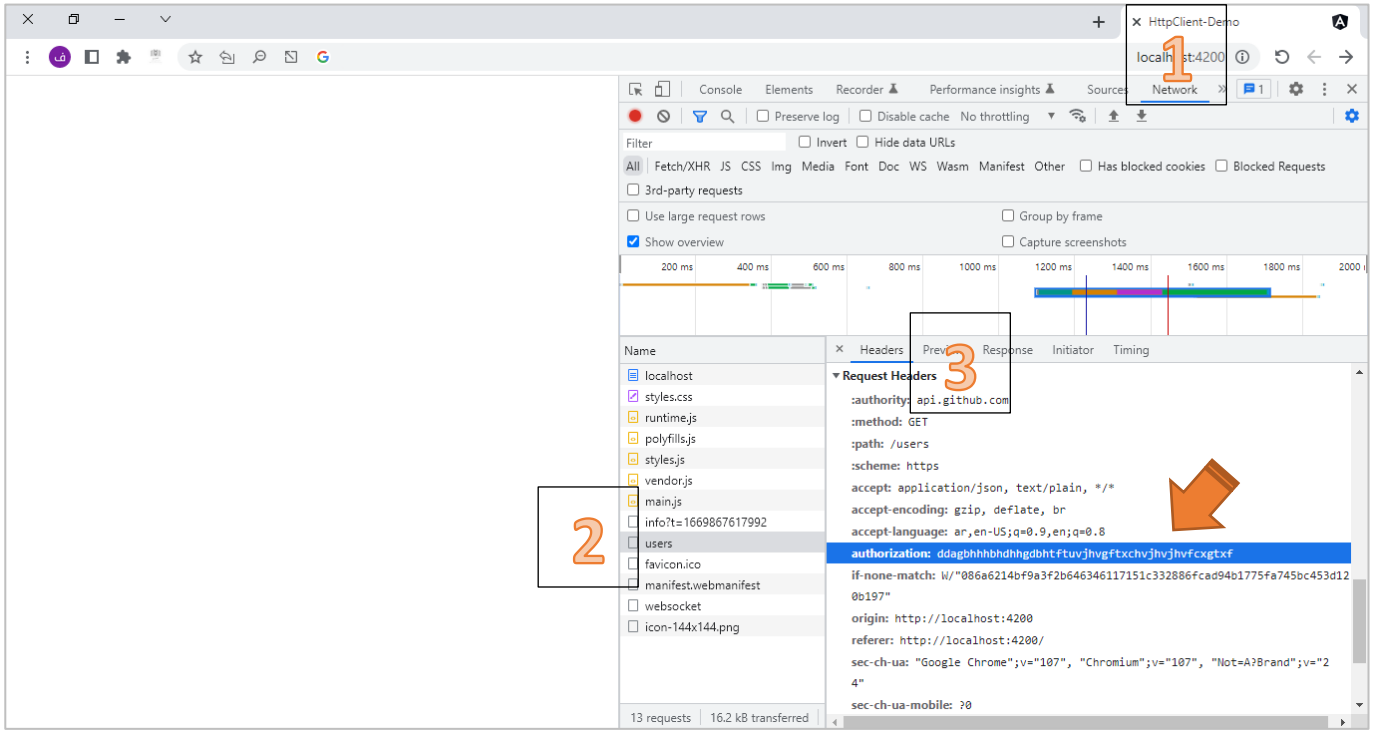
ولو استعرضنا اول دالتين وهما ()set و()append يتم استخدامهما لإضافة خصائص جديدة مع قيمها، والفرق بينهما ان الدالة ()set تبحث في البداية عن الخاصية فإذا وجدتها قامت بتغيير القيمة لها بالقيمة الجديدة وفي حال لم تجدها تقوم بإضافة الخاصية وقيمتها، اما الدالة ()append تقوم بإضافة الخاصية للHeader بغض النظر هل هذه الخاصية موجودة من عدمه.

وفي الحقيقة هاتين الدالتين يتم استخدامهما في إضافة بيانات في الHeader الخاص بالRequest فقط، ومن وجهة نظري أرى ان استخدام الطريقة المذكورة في المثال السابق أفضل ولكن سوف اذكرهما هنا من باب الأمانة العملية، لذلك ولتوضيح لنقوم بتغيير المثال السابق واضافة الخاصية Authorization، مع تغيير الAPI إلى (https://api.github.com/users) باستخدام احدى هذه الدوال، كالتالي:

app.component.ts ملف

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    const token = 'ddagbhhhbhdhhgdbhtftuvjhvgftxchvjhvjhvfcxgtxf';
    const header = new HttpHeaders().append('Authorization', token);
    this.http
      .get('https://api.github.com/users', {
        observe: 'response',
        headers: header,
      })
      .subscribe({
        next: (data) => console.log(data),
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

ولنشاهد النتيجة في المتصفح، وذلك من خلال الذهاب إلى تبويب network ومن ثم اختيار الملف users واخيراً الانتقال إلى الجزء الخاص بـRequest Headers، كالتالي:



كما نلاحظ الخاصية Authorization مع قيمتها تم اضافتها للHeader الخاص بـRequest.

اما باقي الدوال فيتم استخدامهما في الHeader الخاص بالResponse، كأن نستعرض خاصية معينة او قيمتها وهكذا، ما عدا الخاصية delete والتي في الحقيقة لم أجد لها أي استخدام في أرض الواقع.

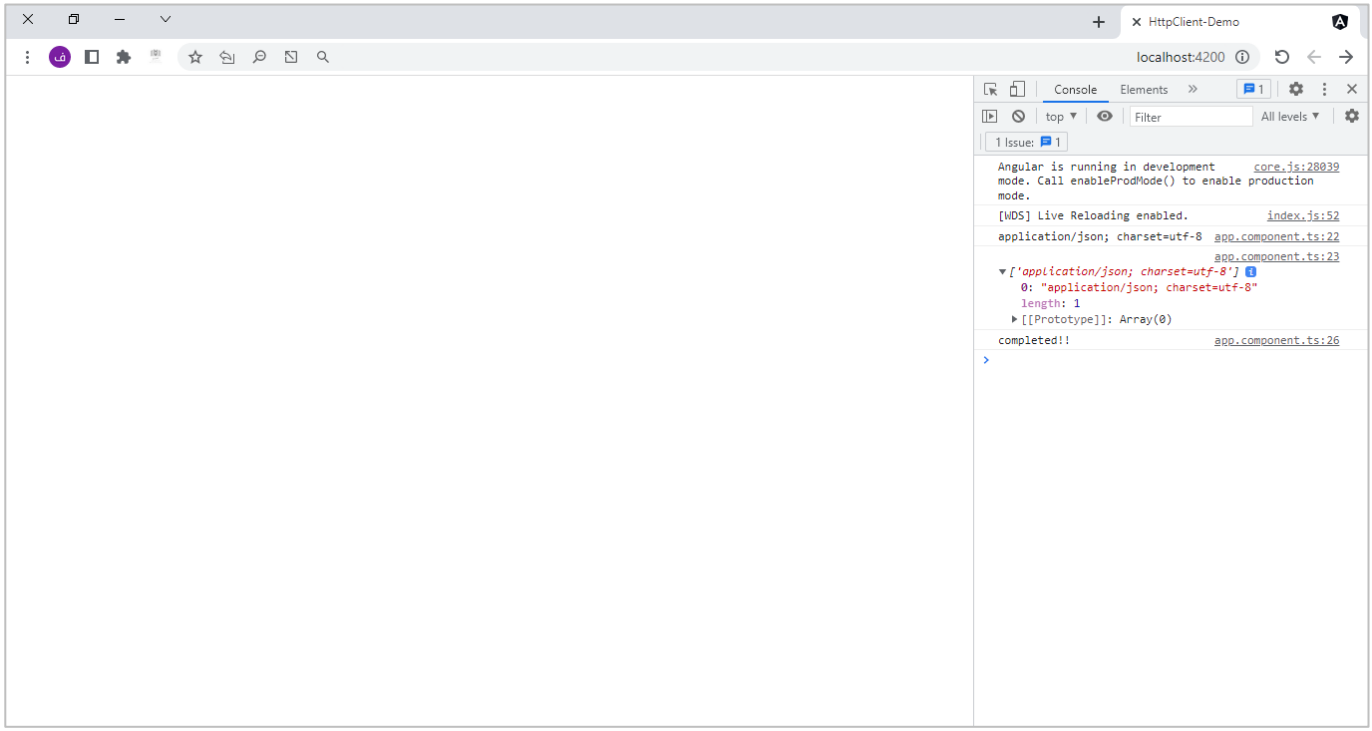
بحيث الدالتين ()get و()getAll فتقومان بجلب القيم لخاصية معينة، حيث نمرر الخاصية لهما وتُعيدان القيمة، اما الفرق بينهما فإن الدالة ()get تُعيد قيمة وحيدة أما ()getAll فتُعيد جميع القيم للخاصية على شكل مصفوفة نصية طبعاً في حال كانت هذه الخاصية تحمل اكثر من قيمة، ولتوضيح لنعطي المثال التالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { map } from 'rxjs/operators';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get('https://api.github.com/users', {
        observe: 'response',
        headers: {
          Authorization: 'ddagbhhhbhdhhgdbhtftuvjhvgftxchvjhvjhvfcxgtxf',
        },
      })
      .pipe(map(({ headers }) => headers))
      .subscribe({
        next: (header) => {
          console.log(header.get('content-type'));
          console.log(header.getAll('content-type'));
        },
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

كما نلاحظ قمنا بطلب طباعة قيم الخاصية content-type من الHeader الخاص بـResponse

ولنشاهد النتيجة في المتصفح، كالتالي:



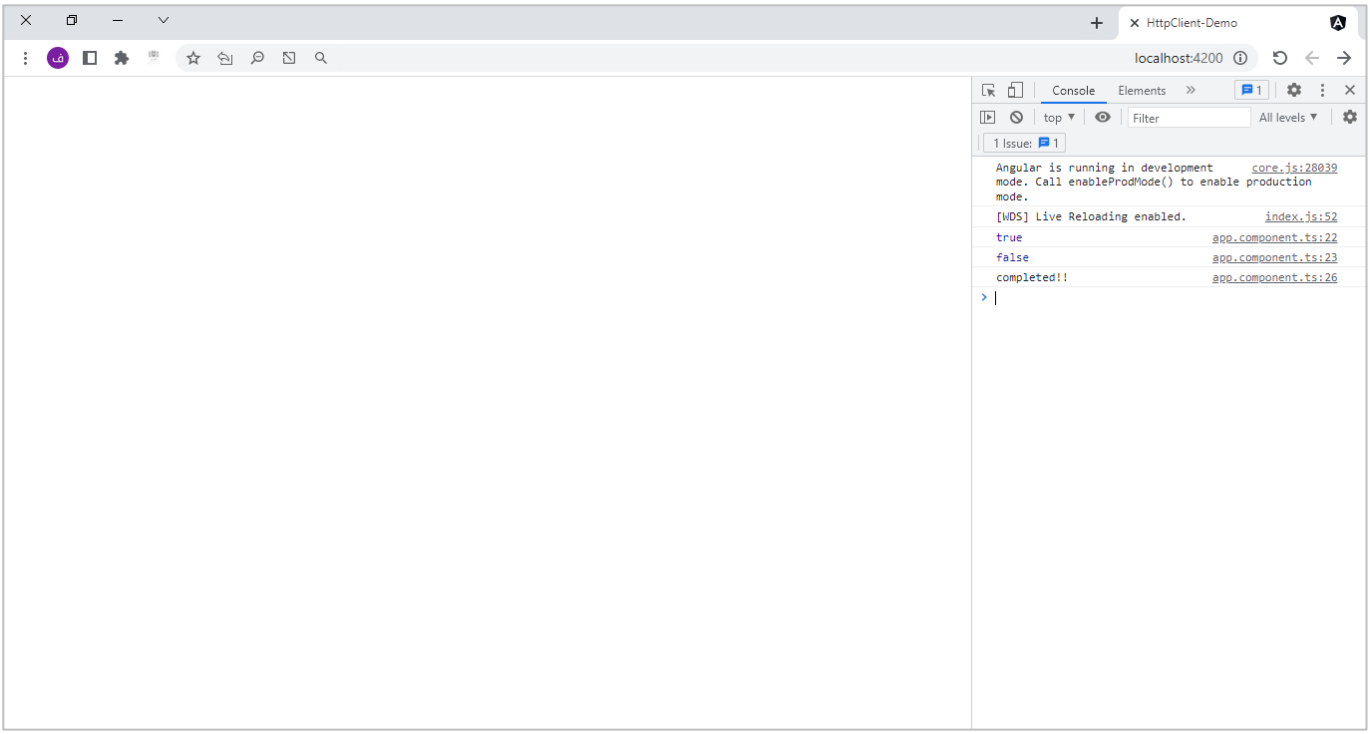
كما نلاحظ ظهرت لنا نتيجتين الأولى لدالة get واظهرت قيمة الخاصية content-type والتي هي قيمة نصية، اما الدالة getAll قامت بإعادة القيم على شكل مصفوفة نصية وبما انه لا توجد الا قيمة واحدة فلذلك ظهرت لنا النتيجة مصفوفة نصية تحتوي على عنصر واحد.

والدالة التالية هي الدالة ()has وهذه الدالة نمرر لها اسم الخاصية وتُعيد لنا قيمة منطقية بحيث تكون true في حال كانت الخاصية موجودة في الheader وتُعيد القيمة false في حال لم تكن موجودة، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { map } from 'rxjs/operators';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get('https://api.github.com/users', {
        observe: 'response',
        headers: {
          Authorization: 'ddagbhhhbhdhhgdbhtftuvjhvgftxchvjhvjhvfcxgtxf',
        },
      })
      .pipe(map(({ headers }) => headers))
      .subscribe({
        next: (header) => {
```

```
          console.log(header.has('content-type'));
          console.log(header.has(' access-control-allow-origin'));

        },
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

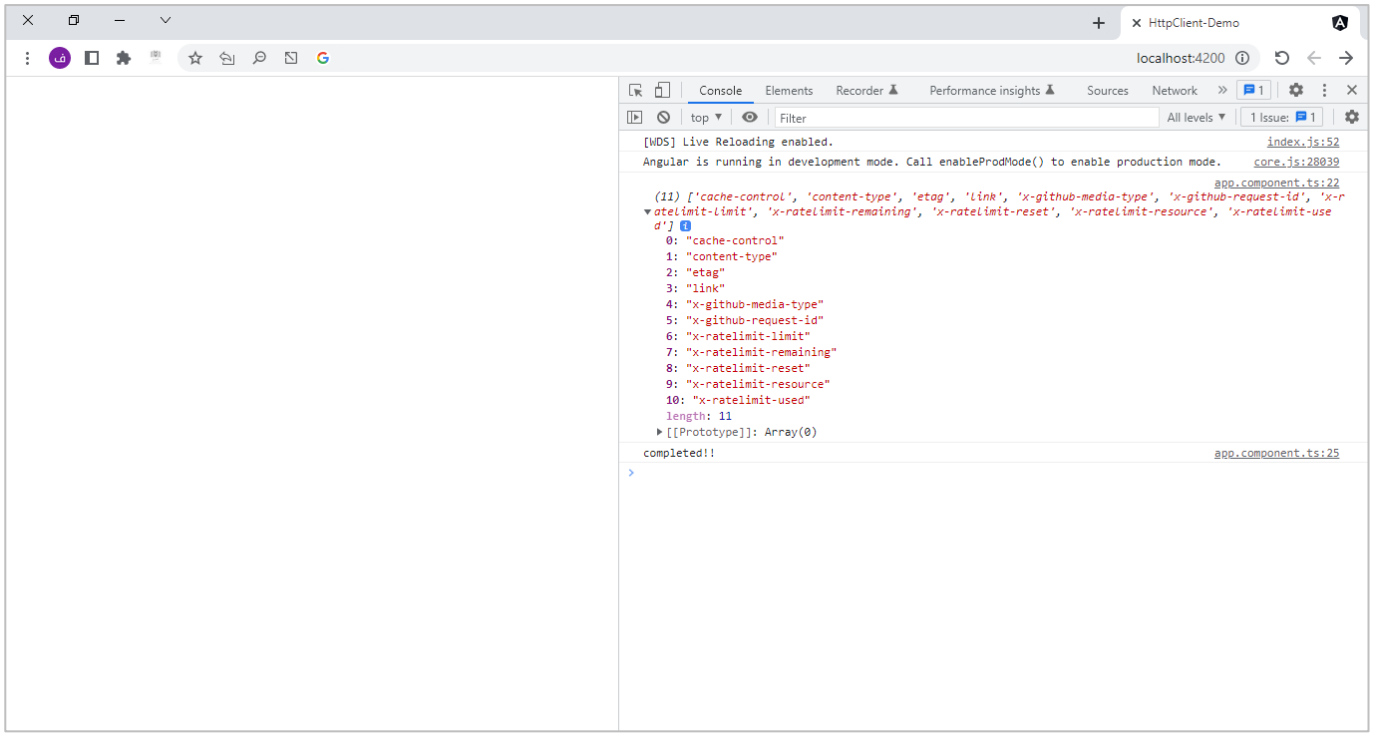


كما نلاحظ الخاصية الأولى true بمعنى موجودة في header اما الأخرى false.

اما آخر دالة لدينا هي keys وتقوم بإرجاع جميع الخصائص في الHeader بدون قيمها – فقط اسم الخاصية – على شكل مصفوفة نصية، كالتالي:

ملفapp.component.ts
```
import { HttpClient } from '@angular/common/http';

import { Component, OnInit } from '@angular/core';
import { map } from 'rxjs/operators';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get('https://api.github.com/users', {
        observe: 'response',
        headers: {
```

```
          Authorization: 'ddagbhhhbhdhhgdbhtftuvjhvgftxchvjhvjhvfcxgtxf',
        },
      })
      .pipe(map(({ headers }) => headers))
      .subscribe({
        next: (header) => {
          console.log(header.keys());
        },
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```

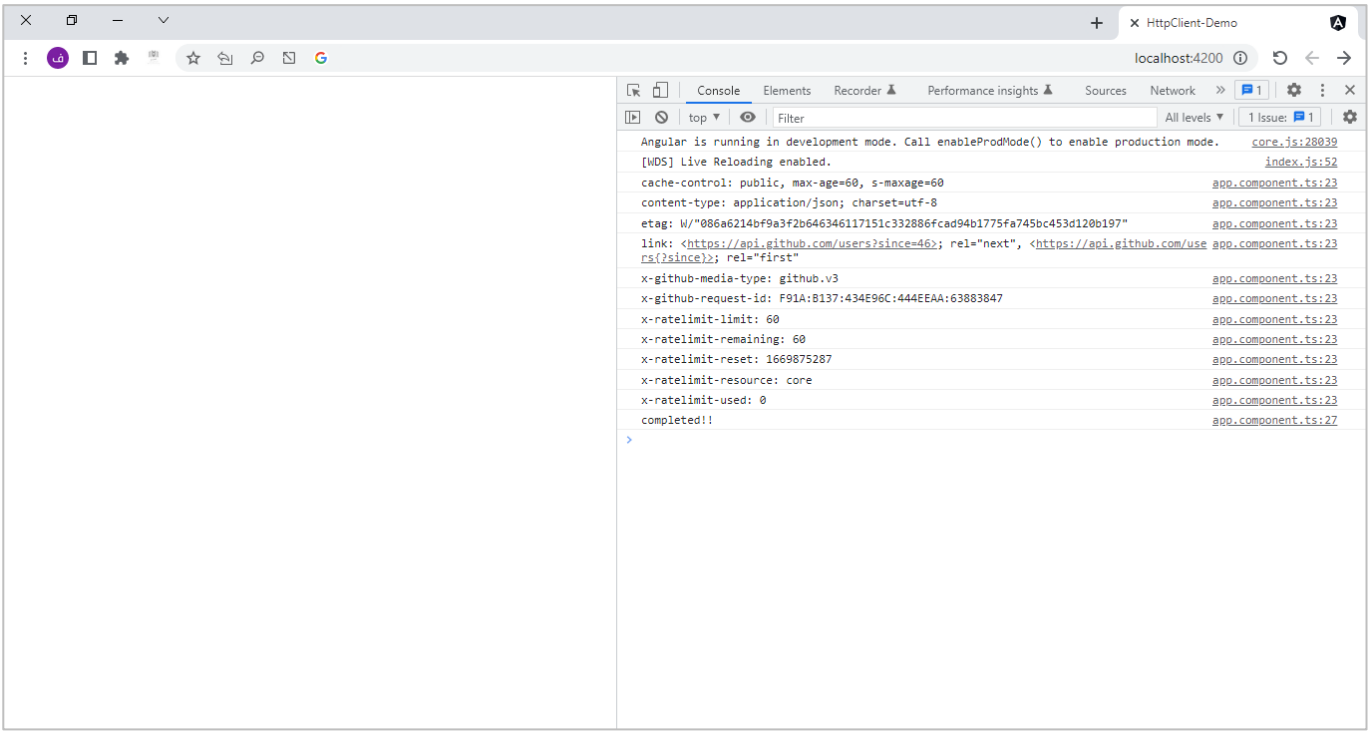


كما نستطيع اظهار الخاصية وقيمتها وذلك من خلال عمل Loop على مصفوفة الخصائص باستخدام الدالة map الخاصة بالمصفوفات ومن ثم نستخدم الدالة get ونمرر لها الخاصية لكي نستخرج القيمة لهذه الخاصية، كالتالي:

ملف app.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { map } from 'rxjs/operators';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http
      .get('https://api.github.com/users', {
```

```
        observe: 'response',
        headers: {
          Authorization: 'ddagbhhhbhdhhgdbhtftuvjhvgftxchvjhvjhvfcxgtxf',
        },
      })
      .pipe(map(({ headers }) => headers))
      .subscribe({
        next: (header) => {
          header.keys().map((key) => {
            console.log(`${key}: ${header.get(key)}`);
          });
        },
        error: (error) => console.log(error),
        complete: () => console.log('completed!!'),
      });
  }
}
```



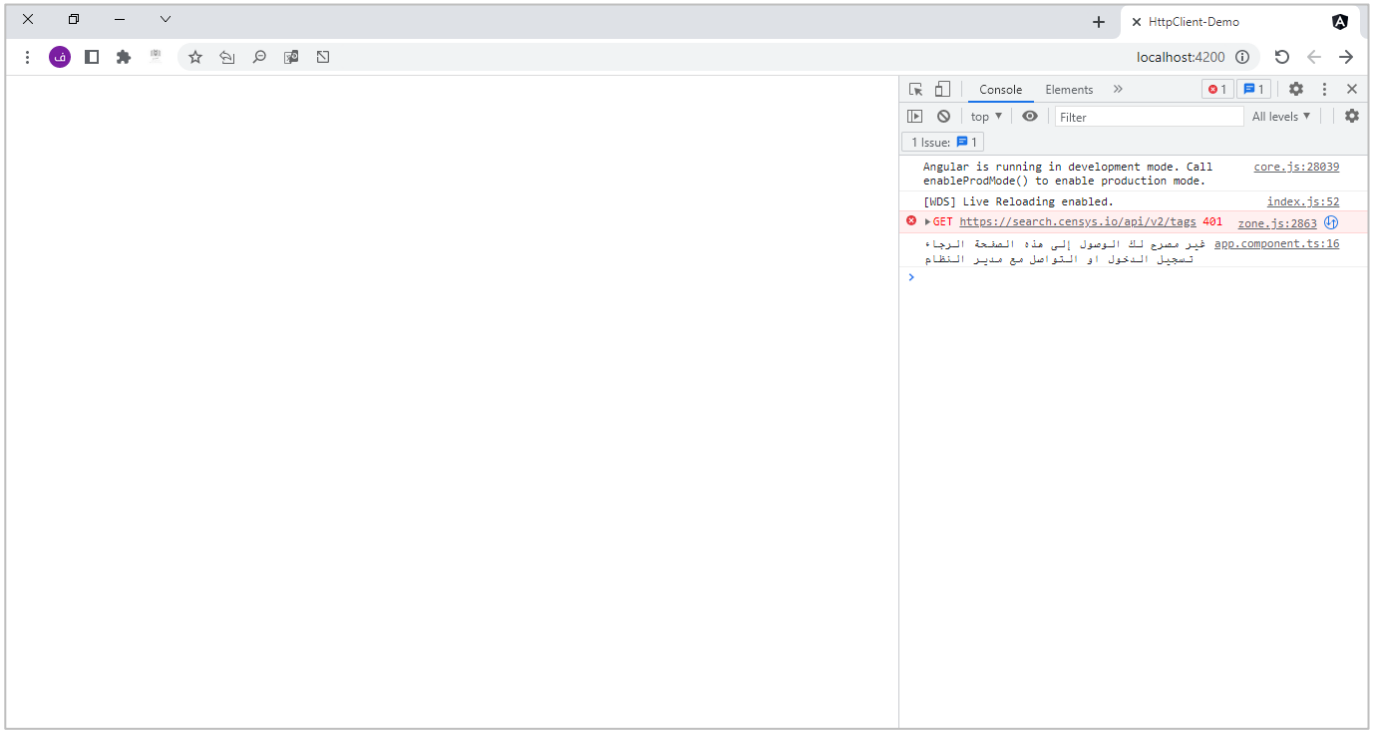
8- معالجة الإخطاء Handle Errors:

كما كنا نتعامل مع الأخطاء في مكتبة RXJS، هي نفسها نستخدمها هنا بجميع تقنياتها المختلفة، وما سوف نضيفه هنا هو ان HttpClient Module يوفر class اسمه HttpResponseError ويحتوي على بعض الخصائص التي نستطيع الاستفادة منها لمعرفة معلومات كثيرة عن نوع هذا الخطأ، مثل رقم الخطأ او رسالة الخطأ او عرض وقراءة Header وغيرها من الخصائص، كما في الصورة التالية:

لذلك نستطيع مثلاً وضع شرط بناءً على status أي رقم الخطأ فإذا كان مثلاً 401 نظهر سالة انه غير مصرح له الوصول إلى هذه الصفحة او 404 الصفحة غير موجودة، او إذا كان الرمز يبدأ بالرقم 5XX نظهر رسالة للمستخدم ان هنالك خطأ في نفس الخادم، وهكذا بقية الرموز، كما في المثال التالي:

ملف app.component.ts

```
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}
  ngOnInit() {
    this.http.get('https://search.censys.io/api/v2/tags').subscribe({
      next: (data) => console.log(data),
      error: (error) => {
        if (error instanceof HttpErrorResponse) {
          if (error.status == 401) {
            console.log(
              'غير مصرح لك الوصول إلى هذه الصفحة الرجاء تسجيل الدخول او التواصل مع مدير النظام'
            );
          }
        }
      },
    });
  }
}
```



9- Query Parameters:

كثير من الأحيان نحتاج إلى أرسال باراميترات عن طريق الAPIs للحصول على معلومات متعددة لنفس هذه APIs، وقد مررنا على مثال من هذا النوع في ثنايا هذا الكتاب عندما تكلمنا عن debounceTime، وسوف نُعيد هذا المثال ولكن بأسلوب الAngular.

حيث وفرت Angular HttpClient Module كلاس جاهز باسم HttpParams، بحيث نستطيع ارسال هذه البارامترات بواسطته بكل سهولة وبساطة.

وهنالك عدة طرق لإضافة البارامترات لروابط API، وهي في حقيقة الأمر مشابهه لطرق إضافة الخصائص في الHeader من حيث استخدام الدوال set وappend و ..الخ، والتي تكلمنا عنها سابقاً، ولكي لا نعيد نفس ما قلنا سوف اتطرق فقط لأسهل وابسط طريقة لإضافة البارامترات بدون التفصيل بالطرق المتعددة.

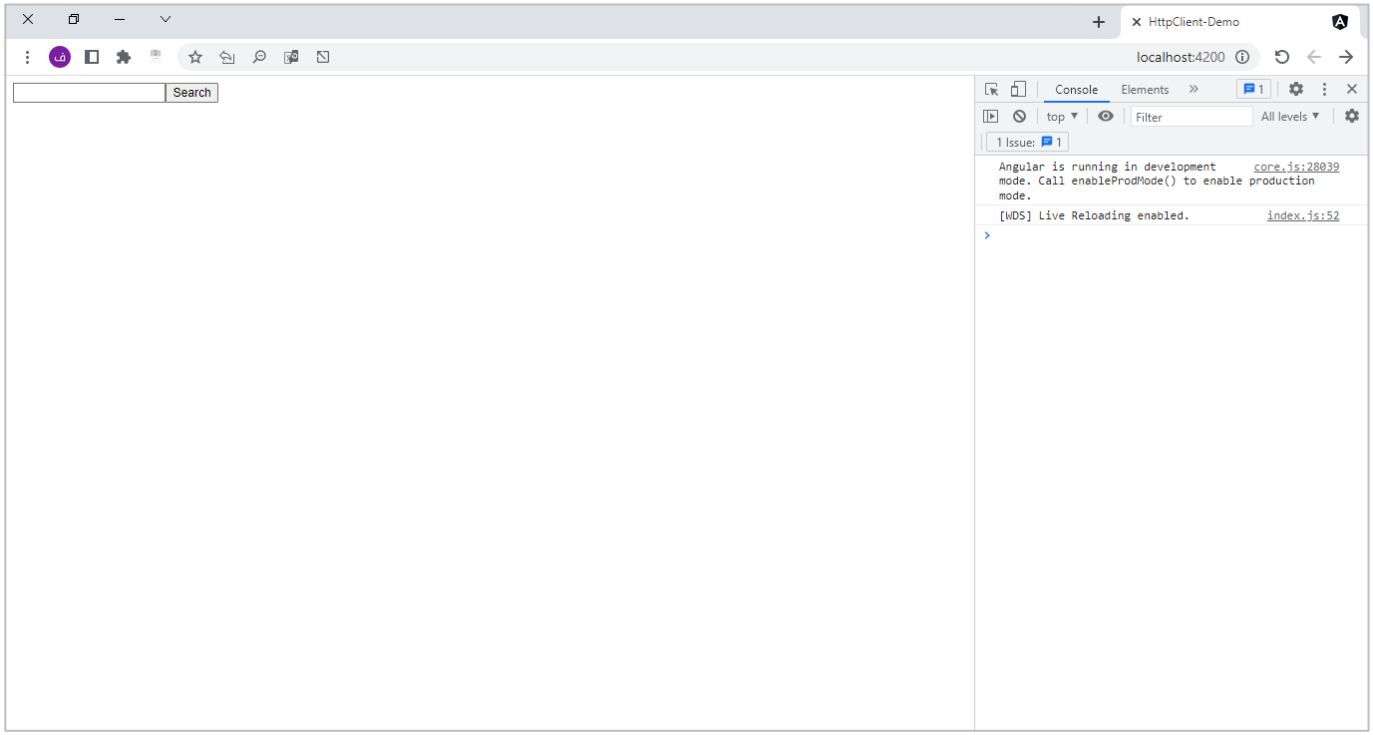
لذلك لنطبق المثال التالي:

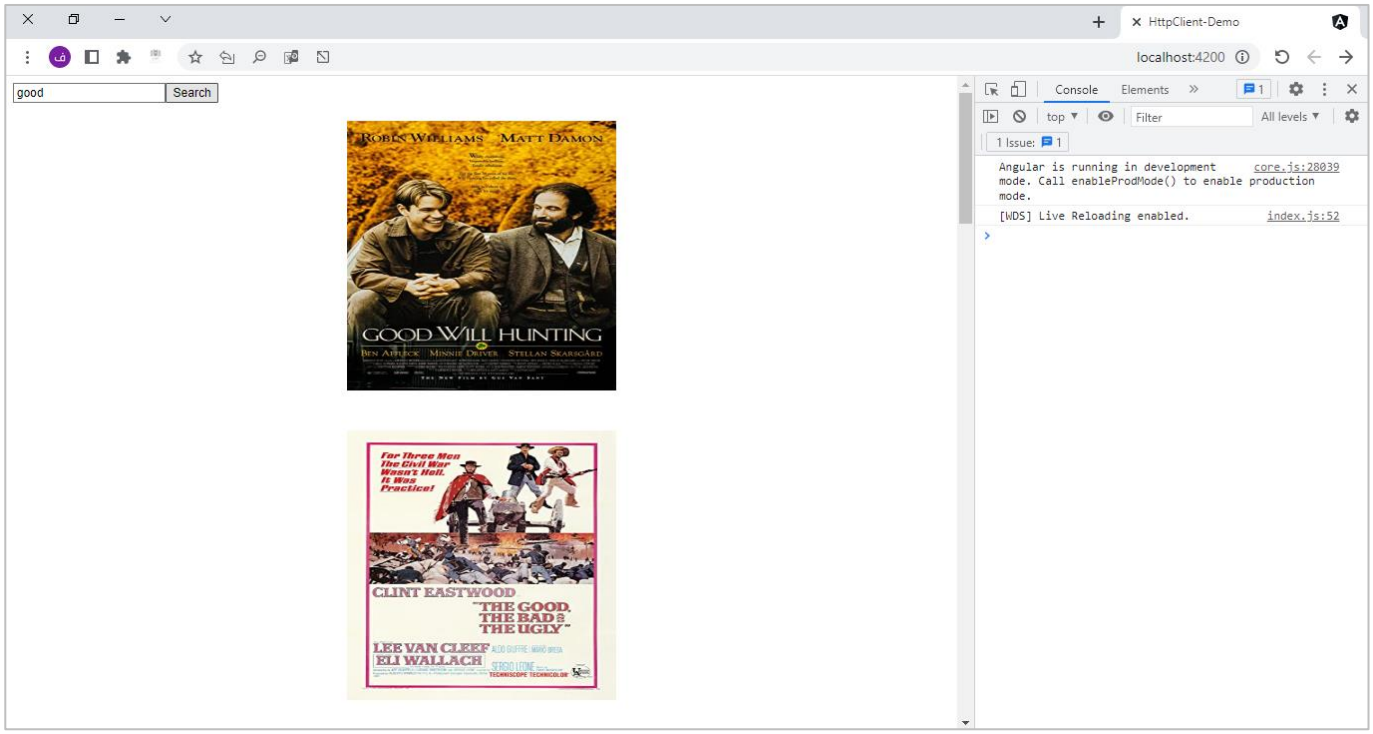
ملف app.component.ts

```
<input type="text" name="search" [(ngModel)]="search_text" />

<input type="button" (click)="onSearch()" value="Search" />

<div *ngFor="let movie of movies" style="text-align: center">
  <img [src]="movie.Poster" width="300" height="300" style="margin: 20px" />
</div>
```

app.component.ts ملف

```ts
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { map } from 'rxjs/operators';

export interface IMovie {
  Poster: string;
  Title: string;
  Type: string;
  Year: string;
  imdbID: string;
  Search: [];
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  search_text: string = '';
  movies: IMovie[] = [];
  constructor(private http: HttpClient) {}
  ngOnInit() {}
  onSearch() {
    this.http
      .get<IMovie>(`http://www.omdbapi.com`, {
        params: {
          apikey: 'e8067b53',
          s: this.search_text,
        },
      })
      .pipe(map((data) => data.Search))
      .subscribe({
        next: (data) => (this.movies = data),
        error: (error) => console.log(error),
      });
  }
}
```





10- Interceptors:

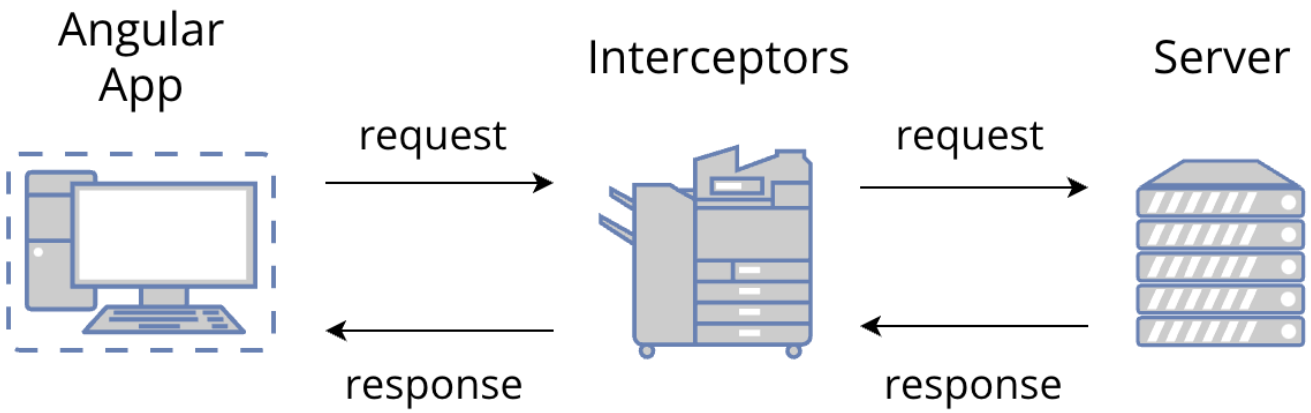
تعتبر ميزة الInterceptors من الميزات المهمة في Angular HttpClient، وهذه الميزة هي ليست تقنية جديدة وانما طريقة لتنفيذ Implement بعض الميزات السابقة الذكر في Angular HttpClient مثل التعامل مع الHeaders او التعامل مع الطلب Request او الاستجابة Response من السيرفر، أو حتى معالجة الأخطاء.

وتتمثل الفكرة الأساسية انه بدلاً من ان نتعامل مع الميزات السابقة في الService او في نفس الComponent (كما كنا نفعل في الأمثلة السابقة) لما لا نجعل هذه المهمة يقوم بها ملف آخر بحيث عندما يبدأ تطبيق Angular بإرسال طلب

Request وقبل ان يتم ارساله إلى السيرفر Server يتم اعتراض هذا الطلب ومن ثم اجراء التعديلات المطلوبة كأن نضيف token، وبنفس الفكرة عند رجوع الاستجابة من الخادم Server (السيرفر) ايضاً يتم اعتراضها عن طريق نفس الملف واجراء التعديلات المطلوبة كمعالجة الأخطاء في حال وجودها.

وهذا الملف هو ما نسميه Interceptor وهو عبارة عن كلاس يعمل Implements للInterface ذات الاسم HttpInterceptor بحيث تحتوي على دالة اسمها intercept وهي التي نكتب بداخلها ما نريد تعديله من ميزات مختلفة.

مع العلم اننا لا نهتم بطريقة اعتراض Request او Response، فكل ما نعمله هو كتابة ما نريد تعديله من ميزات بأسطر قليلة ونترك باقي المهمة للAngular HttpClient، بحيث كلما يتم Request أو Response لاتصال HTTP سيقوم الAngular بتشغيل الملف واعتراض الاتصال وتنفيذ ما يحتويه من تعديل الخصائص التي نُريدها، ونستطيع توضيح مكان الInterceptor في اتصال HTTP من خلال تطبيقات الAngular، كما في الشكل التالي:



مع العمل ان يمكن ان يحتوي مشروع Angular الواحد على أكثر من Interceptor.

واستخدام هذه الميزة المهمة لها فائدة مهمة وهي وضع جميع الLogic البرمجي الخاص بالتعامل مع الHeaders ومعالجة الأخطاء في مكان واحد بحيث يؤدي كل جزء من تطبيق Angular مهامه الخاصة به عند التعامل مع بيانات يتم جلبها بواسطة HTTP، حيث الService تقوم بعمل Request لجلب البيانات ومن ثم استقبال البيانات هذه البيانات من خلال Response والتلاعب بها بحسب الحاجة عن طريق مكتبة RXJS، وارسالها إلى الComponent لعمل Supscription وعرض البيانات للمستخدم، وترك مهمة التعديل على الHeaders ومعالجة الأخطاء للInterceptor، وبذلك نكون حققنا مجموعة من الأهداف منها سهولة التطوير والتعديل على الLogic البرمجي مستقبلاً لأننا قمنا بفصل المهام على حده، وايضاً قللنا من تكرار الشفرة البرمجية، فلو لدينا اكثر من اتصال لAPI معين فأننا نحتاج ان نقوم بإجراء التعديل في كل Logic يقوم بالاتصال بهذا الAPI (كإضافة Token لهذا الاتصال) ولكن باستخدامنا لInterceptors فقد حللنا هذا الامر بحيث كل طلب يتم عن طريق هذا API تلقائياً كود برمجي واحد بملف واحد يقوم باعتراض وتعديل هذا الاتصال ولو اتى من اكثر من جزء من التطبيق.

واخيراً يجدر الإشارة انه يمكن استخدام الInterceptors في عدة طرق، وهي:

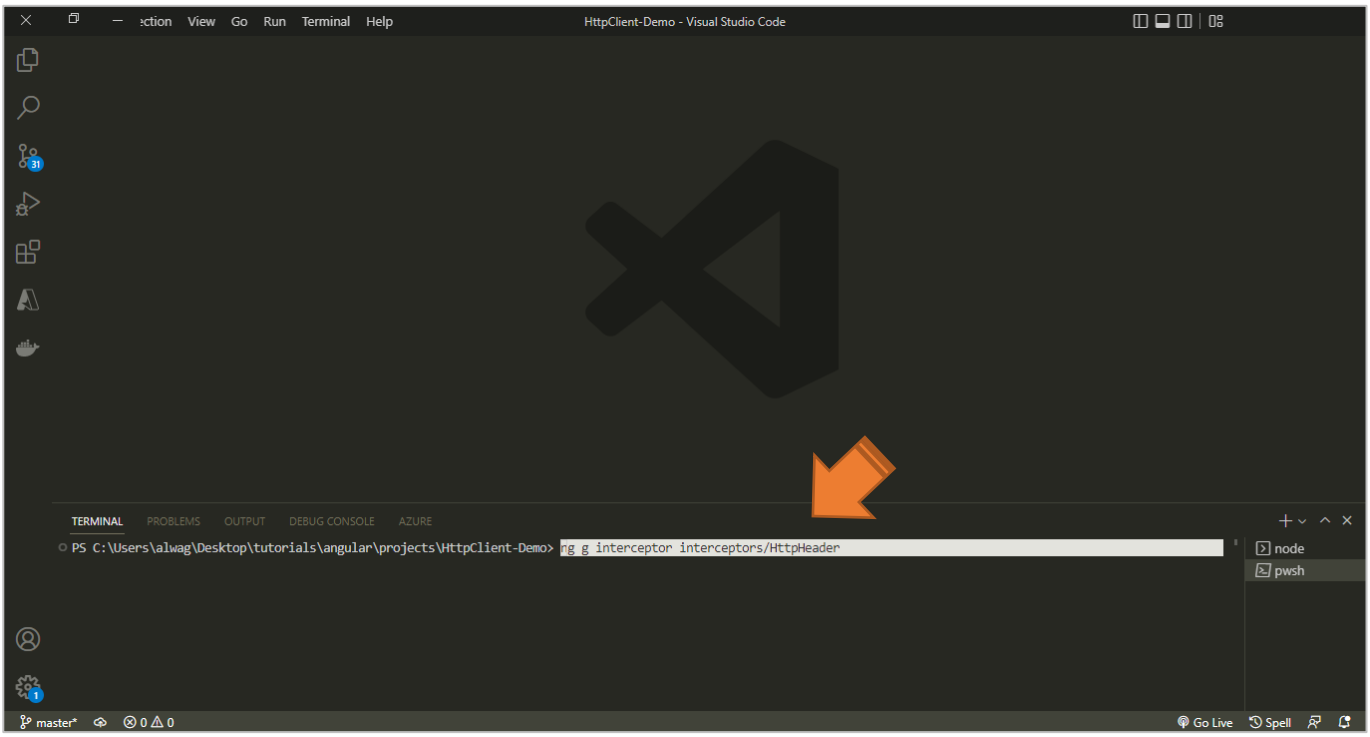
١) التعديل على الHeaders، واضافة الخصائص والTokens في Request قبل إرساله إلى الخادم Server (السيرفر)

٢) نستطيع ايضاً التعديل على الBody (البيانات) في الRequest قبل ارساله إلى الخادم.

٣) التعديل والتحكم بالResponse قبل وصوله إلى الServices او الComponents – (التعديل يكون على Body فقط مع إمكانية قراءة الHeader)

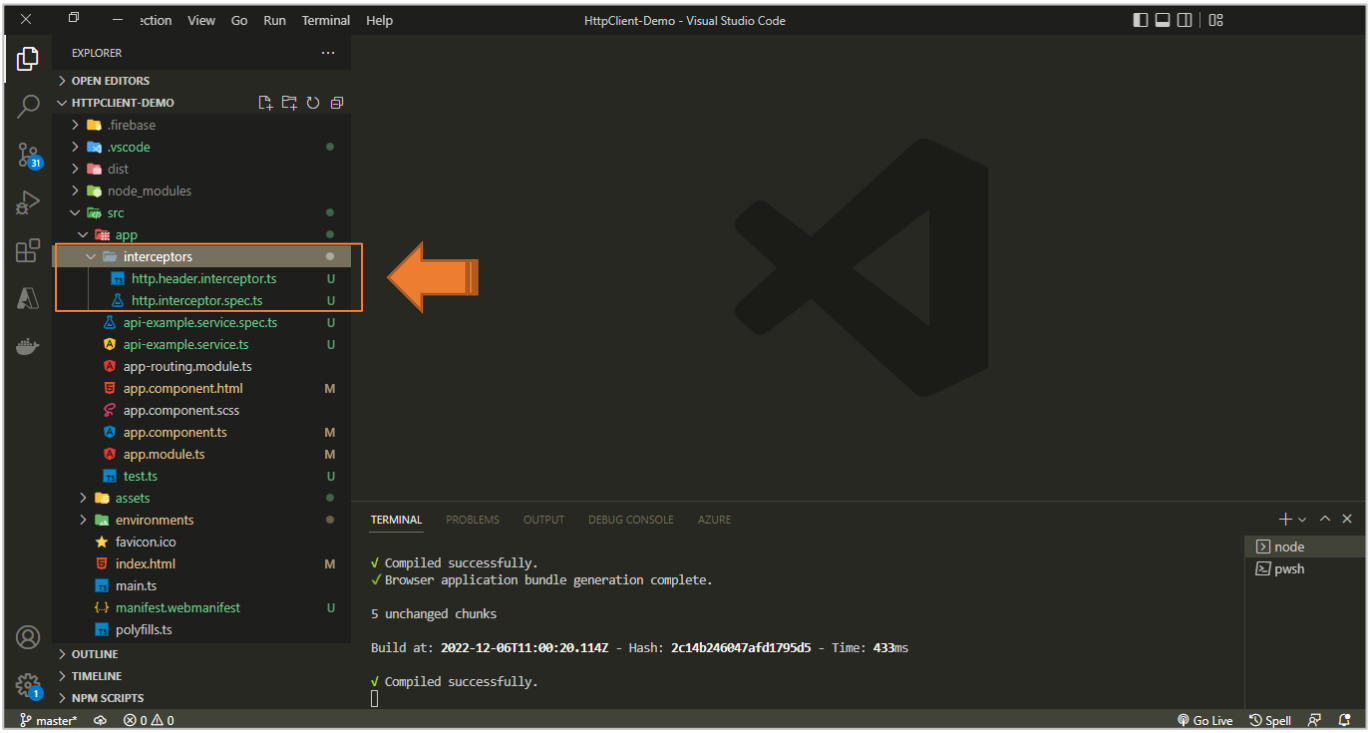
٤) التقاط ومعالجة الأخطاء.

وسوف نتطرق إلى جميع هذه النقاط مع إعطاء مثال لكل منها، ولكن وقبل التطرق إليها لنتعرف على كيفية إضافة Interceptor جديد إلى مشروع الAngular لذلك لنذهب إلى مشروعنا الذي انشئناه سابقاً، ونفتح الTerminal ونكتب الأمر التالي: ng g interceptor ومن ثم نكتب اسم ملف interceptor، كالتالي:



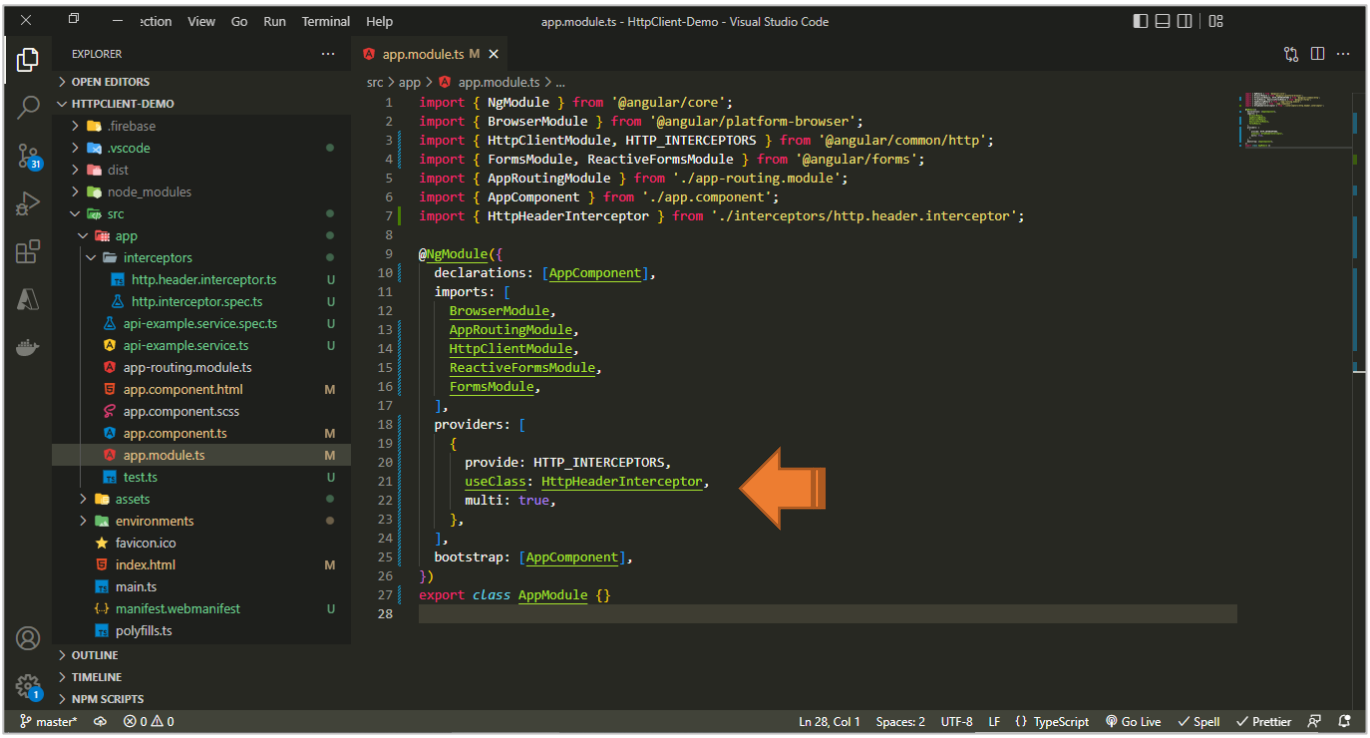
كما نلاحظ كتبنا الامر ng g interceptor interceptors/http ومعناه انشأ مجلد باسم interceptors بداخله ملف Interceptor والذي اسمه HttpHeader، والفائدة من انشاء مجلد بحيث يكون حاوية لأي Interceptor جديد كنوع من التنظيم فقط لاغير.

اما الآن نضغط على زر Enter من لوحة المفاتيح لكي يتم تنفيذ هذا الامر، وعند تنفيذ الامر السابق بنجاح سوف تظهر لنا الملفات التالية:



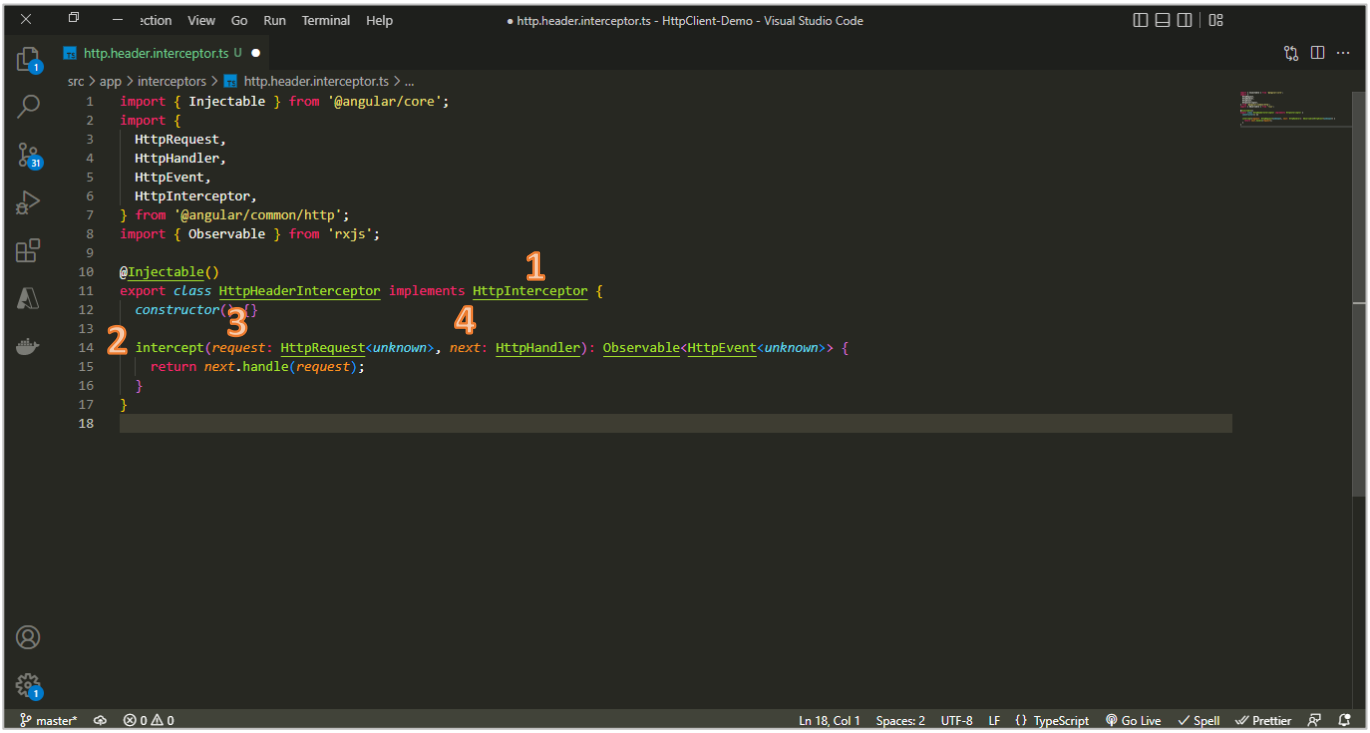
كما نلاحظ تم انشاء ملفين، الملف الأول http.header.interceptor.ts وهو الذي يهمنا اما الآخر فهو لإجراء Testing.

والخطوة التالية هي تسجيل هذا Interceptor في ملف app.module.ts حيث نضيفه في مصفوفة Providers، كالتالي:



كما نلاحظ اضفنا اسم الكلاس داخل ملف http.header.interceptor.ts الذي انشأناه سابقاً والذي اسمه HttpHeaderInterceptor.

والآن اصبح جاهزاً للاستخدام، ولنتصفح محتوى هذا الملف، كالتالي:



كما هو واضح في (1) قمنا بعمل Implements للInterface ذات الاسم HttpInterceptor وهذا الInterface يحمل دالة واحده اسمها intercept وقد تم كتابة محتوياتها افتراضياً في (2) وهذه الدالة هي التي نكتب بداخلها الLogic البرمجي الذي نُريده لاحقاً، كما ان هذه الدالة تحتوي بارامترين وهما request من النوع HttpRequest لكي نستطيع اعتراض أي اتصال Http لاحقاً عن طريقه كما في (3) اما البارامتر الثاني في (4) فنستطيع عن طريقه تنفيذ هذا request كما يمكن تنفيذ اكثر من request لنفس الInterceptor في حال أردنا ذلك، وهذه الدالة تُعيد Observable من النوع HttpEvent.

اما الآن وبعدما استعرضنا كيفية انشاء الInterceptor وما يحتويه، سنقوم الآن بمناقشة النقاط الأربع سابقة الذكر، مع الأمثلة والتوضيح، كالتالي:

10-1- التعديل على الHeaders وإضافة Tokens من خلال Interceptor:

سوف نتطرق في هذا لجزء لكيفية التعديل على الHeader من خلال إضافة أو تعديل خصائص معينة وايضاً سنتطرق لإضافة Tokens لأهميتها وكثرة استخدامها في ميزة الInterceptor، ولإضافة خصائص للHeader سوف نستخدم الAPI التالي (https://free-to-play-games-database.p.rapidapi.com/api/games) ، لذلك لنضيف Service عن طريق كتابة الأمر ng g s ومن ثم اسم الService، وليكن اسمها API، ومن ثم نعمل Inject للكلاس ذو الاسم HttpClient، كالتالي:

api.service.ts ملف

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
```

```
  constructor(private http: HttpClient) {}
}
```

اما الان لننشأ دالة وليكن اسمها getAllGames ونكتب محتواها وهو عمل اتصال Http من نوع get لجلب البيانات عن طريق الAPI (https://free-to-play-games-database.p.rapidapi.com/api/games) ومن ثم نعمل return لهذا الاتصال، كالتالي:

ملف api.service.ts

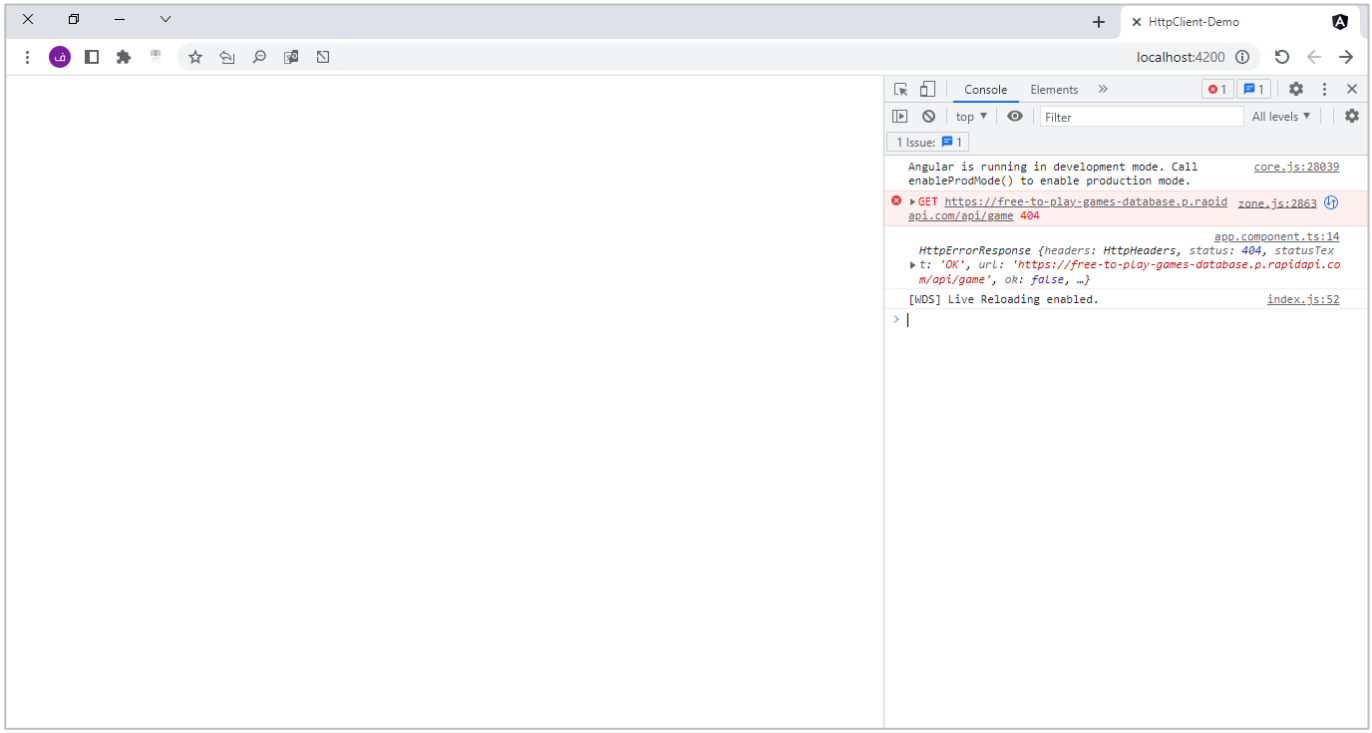
```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  constructor(private http: HttpClient) {}

  getAllGames() {
    return this.http.get(
      'https://free-to-play-games-database.p.rapidapi.com/api/games'
    );
  }
}
```

بعدما قمنا بتجهيز الدالة لنستدعيها في الComponent، بعدما نعمل Inject لهذا الService، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from './services/api.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private api: ApiService) {}
  ngOnInit() {
    this.api.getAllGames().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
  }
}
```

اما الآن لنقوم بتشغيل المشروع ولنرى النتيجة في المتصفح، كالتالي:



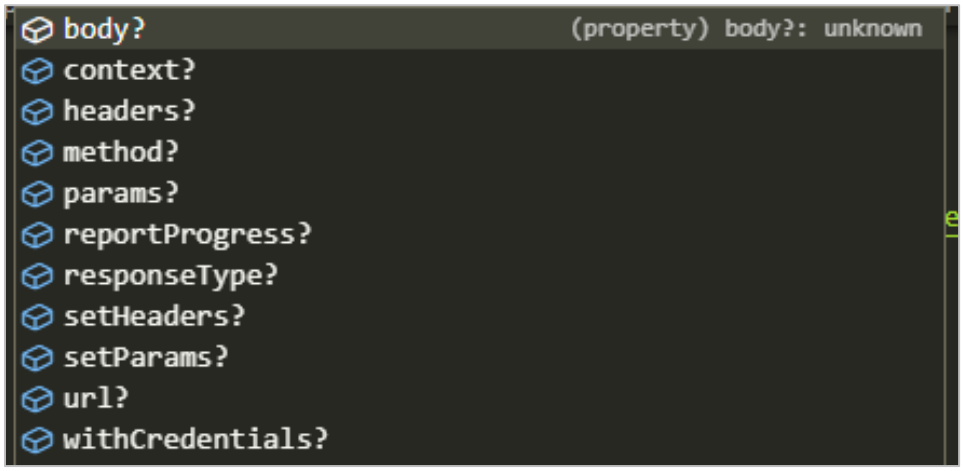
كما نلاحظ ظهر لنا خطأ وهو ان هذا الAPI يحتاج ان نضيف له بعض الخصائص في الHeader لكي يعمل، لذلك اما نضيف هذه الخصائص مباشرة في دالة get كما كنا نفعل سابقاً او نضيفها باستخدام الInterceptor، والأفضل اضافتها في الInterceptor في حال اننا قد نستخدم هذا الAPI في أكثر من مكان من التطبيق، اما في حال كنا لا نستخدم هذا الAPI الا في مكان واحد فتستطيع اضافته مباشرة في دوال HTTP، اما هنا فسوف نضيفها في الInterceptor، كالتالي:

ملف http.header.interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
} from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class HttpHeaderInterceptor implements HttpInterceptor {
  constructor() {}

  intercept(
    request: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {

    request = request.clone({
      setHeaders: {
        'X-RapidAPI-Key': 'b52128808dmsh5826403ec30ac21p1b9548jsnfca5769e0b68',
        'X-RapidAPI-Host': 'free-to-play-games-database.p.rapidapi.com',
      },
    });
```

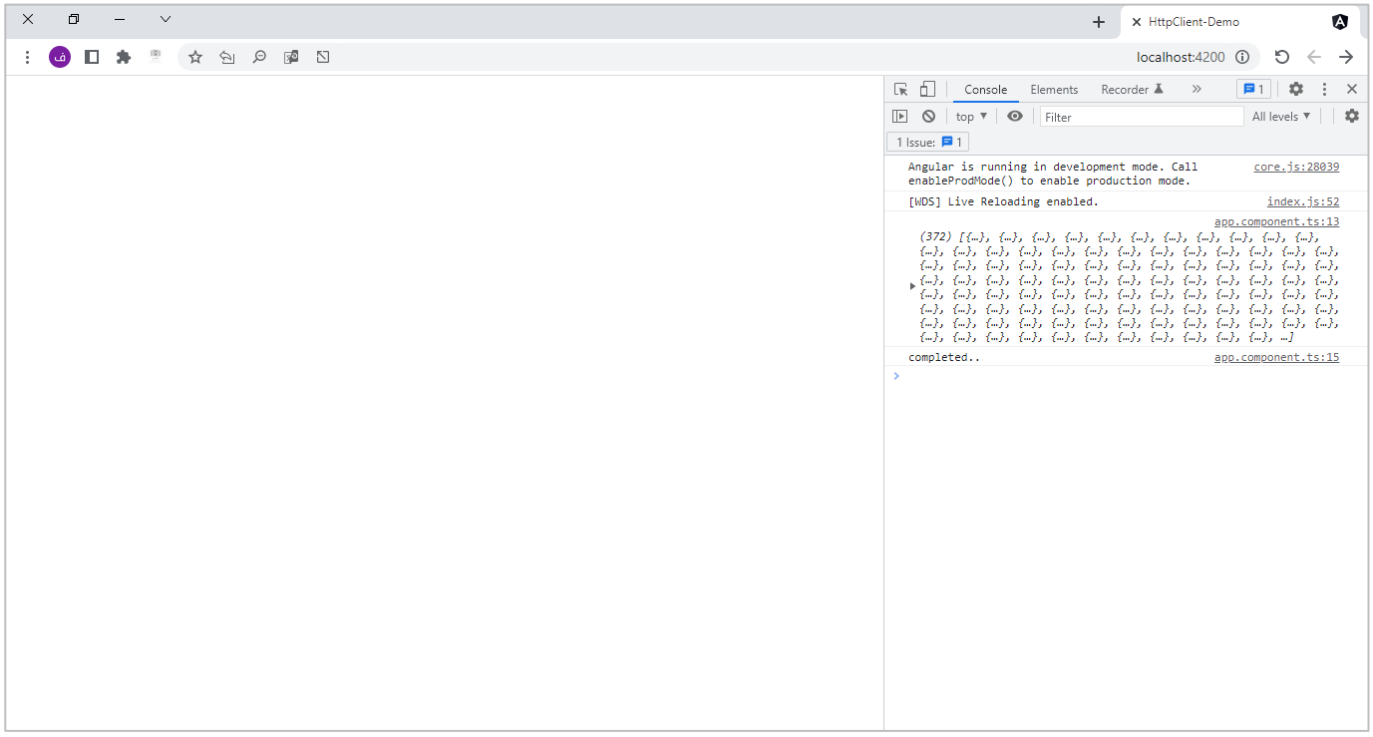
```
    return next.handle(request);
  }
}
```

كما نلاحظ استخدمنا البارامتر request وعن طريق الدالة clone مررنا كائن وفيه استخدمنا الخاصية setHeaders والذي هو الآخر يستقبل كائن وهذا الكائن هو الذي مررنا له الخصائص التي نريد اضافتها إلى هذا الـHeader، كما تجدر الإشارة ان الدالة clone نستطيع ان نمرر لها مجموعة أخرى من الخصائص، كما في الشكل التالي:



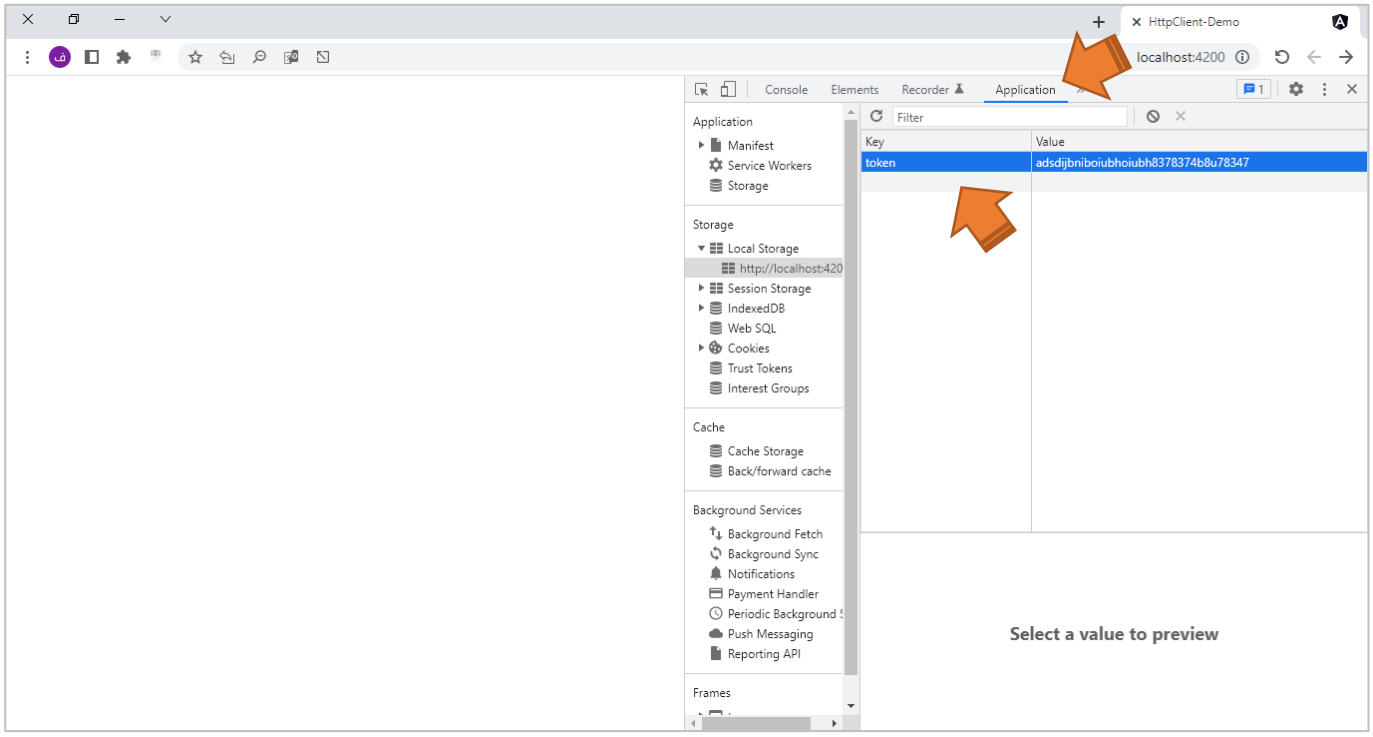
فنستطيع مثلاً إضافة الـHeader باستخدام الخاصية header كما كنا نفعل سابقاً في الأمثلة السابقة بدلاً من setHeaders، كما نستطيع ايضاً تغيير رابط الـAPI لأي سبب من الأسباب او تغيير دالة الـhttp كأن تكون من GET إلى POST، او التعديل على Body، وفق شروط معينة، وهكذا بقية الخصائص الأخرى.

اما ألان لنشاهد النتيجة في المتصفح، كالتالي:



كما نلاحظ تم جلب البيانات بشكل صحيح.

اما الآن لنعرف كيفية إضافة الToken لرابط الAPI، وفي هذه الحالة سوف نستخدم رابط API آخر وليكن (https://jsonplaceholder.typicode.com/users)، ولكن هنا كنوع من التدريب لنفرض ان الToken موجود في ملف آخر وليكن اسمه shared.ts بحيث يحتوي على ثابت نخزن فيه قيمة هذا Token من الLocal Storage، لذلك لنقوم بالذهاب إلى الLocal Storage في المتصفح وإضافة خاصية باسم token وقيمتها عشوائية، كالتالي:



الآن لننشأ ملف shared.ts ولنضيف فيه الثابت token ونسند له القيمة القادمة لنا من Local Storage، كالتالي:

ملف shared.ts

```
export const token = localStorage.getItem('token');
```

اما الآن لنذهب إلى ملف الService ونضيف دالة وليكن اسمها getAllUsers، كالتالي:

ملف api.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  constructor(private http: HttpClient) {}

  getAllGames() {
    return this.http.get(
      'https://free-to-play-games-database.p.rapidapi.com/api/games'
    );
```

```
  }
  getAllUsers() {
    return this.http.get('https://jsonplaceholder.typicode.com/users');
  }
}
```

وقبل استدعاء هذه الدالة في الComponent لنضيف Interceptor جديد بحيث يهتم بإضافة الtoken لهذه الAPI، وليكن اسمه HttpHeader2، ونضيف له الLogic الخاص بإضافة الToken إلى الHeader، كالتالي:

ملف http.header2.interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { TOKEN } from '../shared';

@Injectable()
export class HttpHeader2Interceptor implements HttpInterceptor {

  constructor() {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler):
Observable<HttpEvent<unknown>> {
    request = request.clone({
      setHeaders: {
        token: TOKEN as string,
      },
    });
    return next.handle(request);
  }
}
```

وبطبيعة الحال لا ننسى تسجيل هذا الInterceptor في ملف app.module.ts، كالتالي:

ملف app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpHeaderInterceptor } from './interceptors/http.header.interceptor';
import { HttpHeader2Interceptor } from './interceptors/http-header2.interceptor';

@NgModule({
```

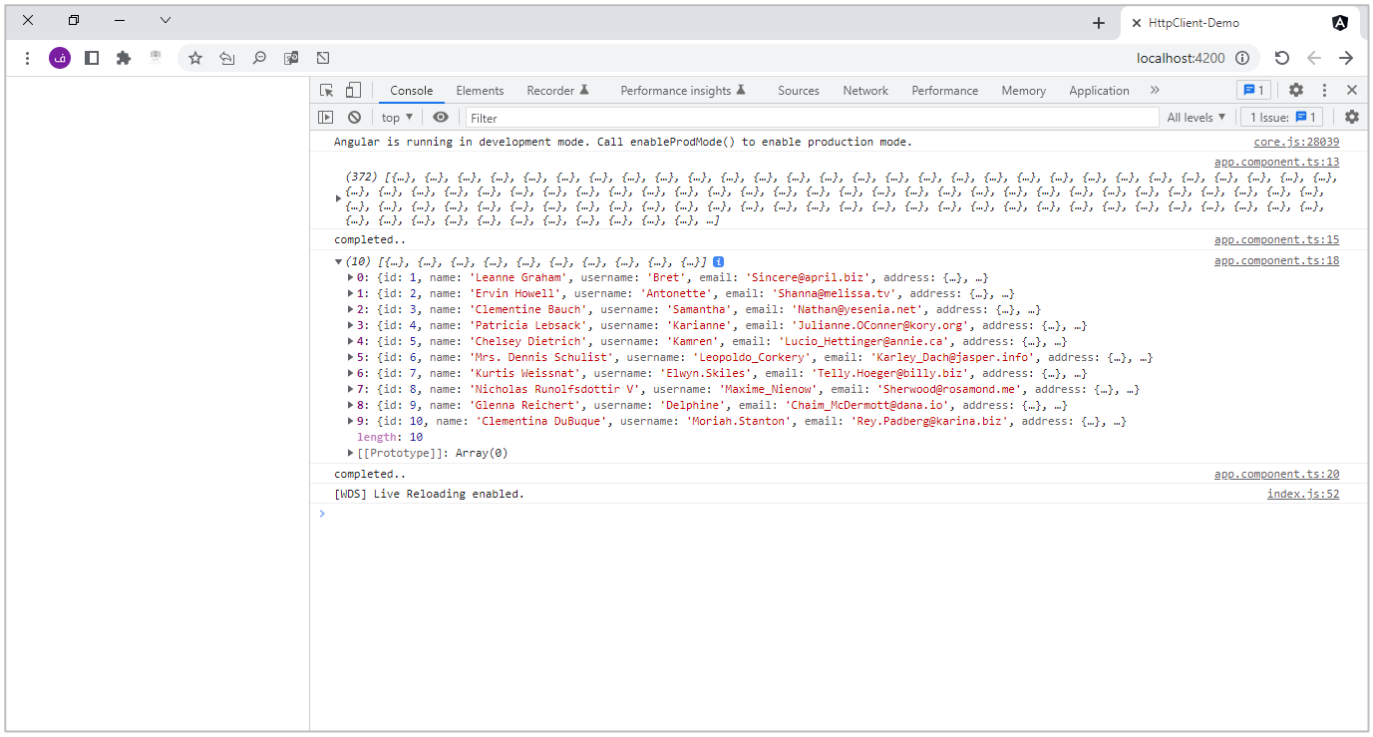
```
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    ReactiveFormsModule,
    FormsModule,
  ],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: HttpHeaderInterceptor,
      multi: true,
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: HttpHeader2Interceptor,
      multi: true,
    },
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

وأخيراً لنقرأ البيانات في الComponent، كالتالي:

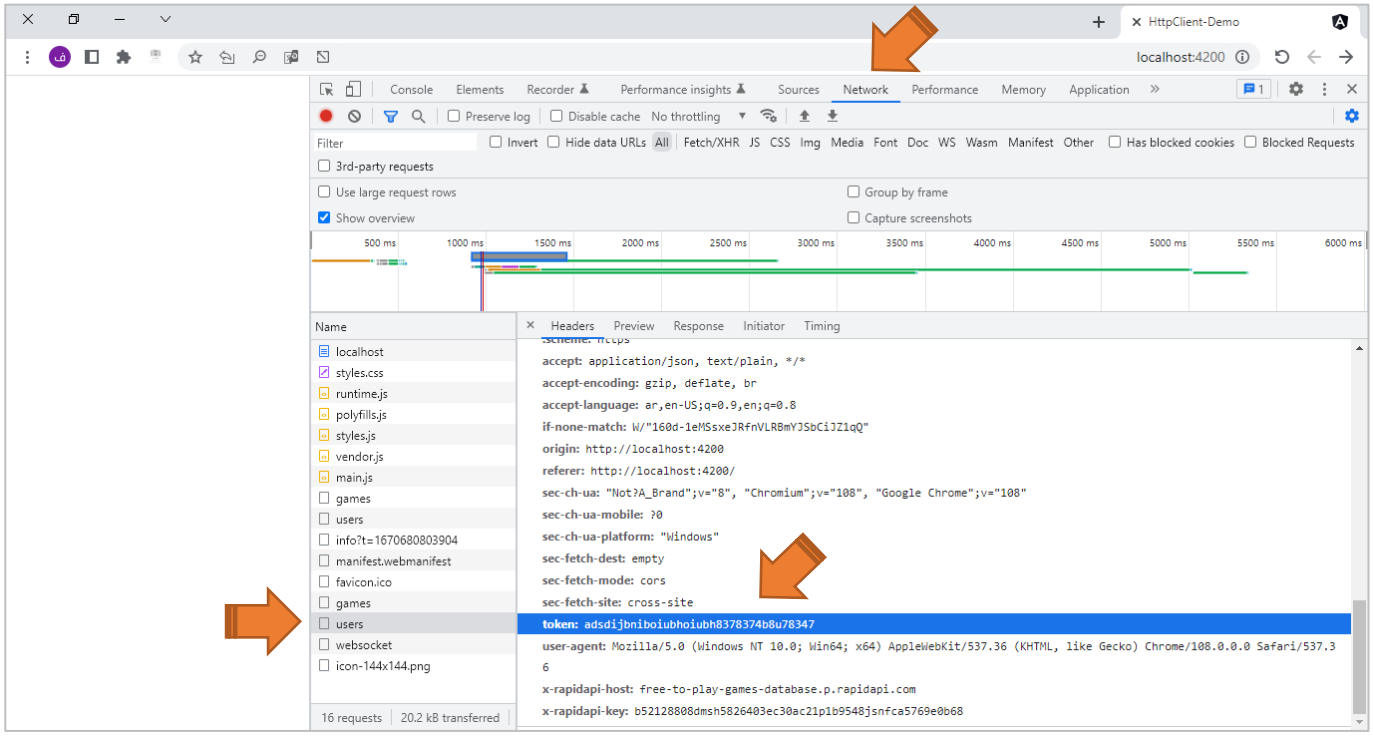
ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from './services/api.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private api: ApiService) {}
  ngOnInit() {
    this.api.getAllGames().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
    this.api.getAllUsers().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
  }
}
```

والآن لنشاهد النتيجة في المتصفح:



وبنفس الوقت لنشاهد ال،Header في Request، وذلك من خلال الذهاب إلى تبويب network ومن ثم اختيار الملف users، كالتالي:



كما تلاحظ عزيزي المتعلم تم إضافة الخاصية token إلى الHeader.

اما الآن لنُزيد الجرعة قليلاً ولنظيف دالة تقرأ رابط API آخر وليكن (https://jsonplaceholder.typicode.com/posts) وسوف نضيفه بنفس الInterceptor الخاص بإضافة token ولكن هنا نريد ان لا يضيف هذا الtoken لهذا الرابط، ولكن

وقبل التطرق لكيفية حل هذا الأمر لنقوم اولاً بإنشاء دالة في الService للاتصال وجلب البيانات وليكن اسمها getAllPosts، كالتالي:

ملف api.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  constructor(private http: HttpClient) {}

  getAllGames() {
    return this.http.get(
      'https://free-to-play-games-database.p.rapidapi.com/api/games'
    );
  }
  getAllUsers() {
    return this.http.get('https://jsonplaceholder.typicode.com/users');
  }
  getAllPosts() {
    return this.http.get('https://jsonplaceholder.typicode.com/posts');
  }
}
```

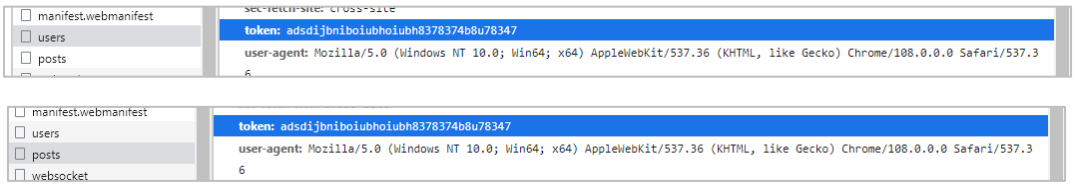
ولنقرأ هذه البيانات في الComponent، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from './services/api.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private api: ApiService) {}
  ngOnInit() {
    this.api.getAllGames().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
    this.api.getAllUsers().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
```

```
    this.api.getAllPosts().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
  }
}
```

والآن لنشاهد النتيجة في المتصفح، كالتالي:



كما تلاحظ عزيزي المتعلم في كلا الاتصالين تم إضافة الtoken، وهذا عكس ما نتوقع وهو إضافته فقط في رابط جلب بيانات الusers فقط، ونستطيع حل هذا الامر بعدة طرق منها استخدام الكلاس HttpContextToken والكلاس HttpContext حيث الأول نستفيد منه بتعريف متغير عام من النوع boolean وذلك عن طريق تمرير دالة كبارامتر لهذا الكلاس وتُعيد قيمة منطقية وفي مثالنا هذا سوف نجعلها مبدئياً تُعيد false، اما الكلاس الثاني فنستفيد منه بتغيير قيمة هذا المتغير العام من خلال إضافة كبيانات إضافية لاتصال Http ونجعل قيمته true في حال اردنا إضافة token لهذا الAPI او false في حالة لم نريد إضافة الtoken للAPI الآخر، بحيث نقوم بهذا الأمر من خلال وضع شرط في الInterceptor، ومع الأمثلة ستتوضح الأمور، لذلك لنذهب إلى ملف shared.ts الذي انشأناه سابقاً ولنعرف متغير جديد من النوع HttpContextToken ولنجعل قيمته مبدئياً false، كالتالي:

ملف shared.ts

```
import { HttpContextToken } from '@angular/common/http';

export const PASS_TOKEN = new HttpContextToken(() => false);
export const TOKEN = localStorage.getItem('token');
```

ومن ثم في ملف api.service.ts نضيف متى نريد إضافة الtoken من عدمه من خلال تغيير قيمة هذا المتغير من true او false، كالتالي:

ملف api.service.ts

```
import { HttpClient, HttpContext } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { PASS_TOKEN } from '../shared';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  constructor(private http: HttpClient) {}
```

```
  getAllGames() {
    return this.http.get(
      'https://free-to-play-games-database.p.rapidapi.com/api/games'
    );
  }
  getAllUsers() {
    return this.http.get('https://jsonplaceholder.typicode.com/users', {
      context: new HttpContext().set(PASS_TOKEN, true),
    });
  }
  getAllPosts() {
    return this.http.get('https://jsonplaceholder.typicode.com/posts', {
      context: new HttpContext().set(PASS_TOKEN, false),
    });
  }
}
```

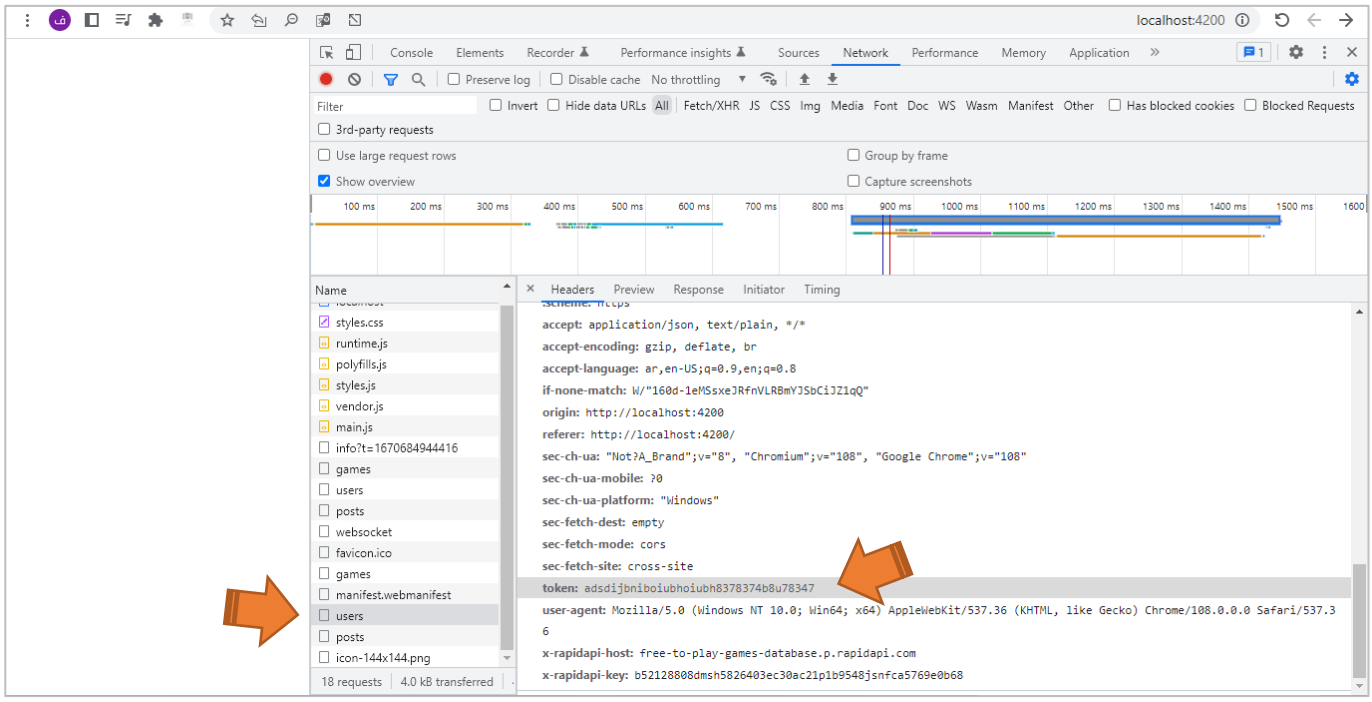
وأخيراً في ملف http.header2.interceptor.ts نقرأ قيمة هذا المتغير وبناءً على القيمة نضيف الtoken ام لا، كالتالي:

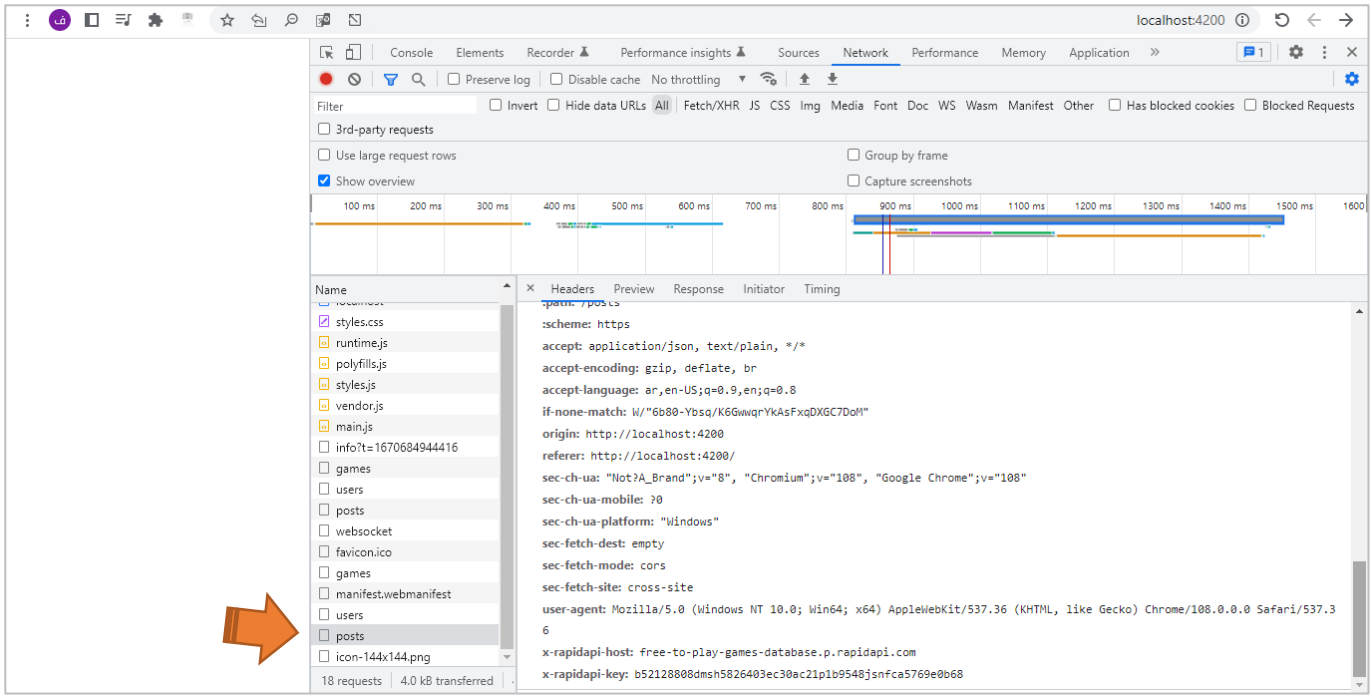
ملف http.header2.interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
} from '@angular/common/http';
import { PASS_TOKEN, TOKEN } from '../shared';

@Injectable()
export class HttpHeader2Interceptor implements HttpInterceptor {
  constructor() {}

  intercept(
    request: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    if (request.context.get(PASS_TOKEN) === false) {
      return next.handle(request);
    } else {
      request = request.clone({
        setHeaders: {
          token: TOKEN as string,
        },
      });
      return next.handle(request);
    }
  }
}
```

كما هو واضح قمنا بقراءة قيمة المتغير من الرابط وفي حال كانت false لا تعمل شيء وفي حال كانت true اضف الtoken.

اما الآن لنشاهد النتيجة في المتصفح من خلال التبويب network، كالتالي:





كما نلاحظ تم إضافة الخاصية token في users ولم يتم اضافتها في posts، وايضاً نلاحظ وجود خاصيتين لا ينتميان لا إلى users ولا إلى posts وهما (x-rapidapi-key - x-rapidapi-host) وهما خاصتنا بالرابط (https://free-to-play-games-database.p.rapidapi.com/api/games) وهو الخاص بالgames والذي ذكرناه في بداية هذا المثال لذلك اترك لك عزيزي المتعلم حل هذا الأمر من خلال عدم اضافته في posts وusers واضافته فقط في games.

10-2- التعديل على الBody في Request من خلال Interceptor:

في بعض الأحيان قد نحتاج إلى ان نقوم بالتعديل على الBody للRequest في حالة إضافة بيانات إلى قاعدة البيانات (POST)، فمثلاً لو كان لدينا اكثر من مكان في تطبيقنا يقوم بإضافة بيانات بنفس الAPI وبنفس الدالة (PUT - POST) ولكن نُريد لأي سبب من الأسباب عند إضافة هذه البيانات ان نقوم بالتعديل عليها وفق شروط معينة قبل أرسالها إلى الServer باستخدام ميزة الInterceptor.

وفي الحقيقة لم اجد مثال من الواقع للحاجة إلى هذه الميزة (وهذا لا يعني انها غير مهمة او ليس لها استخدامات)، لذلك لنعطي مثال وان لم يكن واقعي ولكن لتوضيح هذه الميزة وطريقة استخدامها.

وسوف يكون المثال لدينا زرين، الزر الأول لأضافة مستخدم جديد والزر الثاني لإضافة مستخدم admin بحيث كلا الزرين يقومان بإضافة نفس بُنية البيانات مع قيم مختلفة ولكن في الInterceptor نقوم بعمل شرط بحيث إذا تحقق نضيف بيانات إضافية في حالة كان المستخدم admin او نبقيها على ما هي عليه في حال كان مسنخدم جديد فقط.

لذلك سوف نحتاج إلى متغير عام سوف نضيفه عن طريق context كما تعلمنا سابقاً، ومن ثم نقرأ هذا المتغير وبناءً على قيمته نضيف البيانات الإضافية او لا.

وبناءً على ما سبق سوف نحتاج إلى إضافة interceptor جديد وانا سوف اسميه HttpBodyAdminInterceptor ولا ننسى ان نقوم بتسجيله في AppModule، وايضاً سوف نضيف Interface لتعريف نوع جديد لوصف بنية البيانات التي سوف نضيفها وليكن اسمه IUser، وسوف نضيف هذا الملف داخل مجلد اسمه interfaces.

لذلك اولاً لنقوم بتعريف بنية البيانات، كالتالي:

ملف iuser.interface.ts

```
export interface IUser {
  email: string;
  id: number;
  name: string;
  phone: string;
  username: string;
  website: string;
}
```

وايضاً لنقوم بتعريف متغير عام وليكن أسمه IS_ADMIN في ملف shared.ts، كالتالي:

ملف shared.ts

```
import { HttpContextToken } from '@angular/common/http';

export const PASS_TOKEN = new HttpContextToken(() => false);
export const TOKEN = localStorage.getItem('token');
export const IS_ADMIN = new HttpContextToken(() => false);
```

اما الخطوة التالية ننشأ دالة وليكن اسمها addUser وتستقبل بارامترين الأول هي البيانات التي سوف نضيفها من النوع IUser، والثاني منطقي من النوع boolean ونستفيد منه لكي نمرر القيمة للمتغير IS_ADMIN، كالتالي:

ملف api.service.ts

```
import { HttpClient, HttpContext } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { IUser } from '../interfaces/iuser';
import { IS_ADMIN, PASS_TOKEN } from '../shared';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  constructor(private http: HttpClient) {}

  getAllGames() {
    return this.http.get(
      'https://free-to-play-games-database.p.rapidapi.com/api/games'
    );
  }
  getAllUsers() {
    return this.http.get<IUser[]>(
      'https://jsonplaceholder.typicode.com/users',
      {
        context: new HttpContext().set(PASS_TOKEN, true),
      }
    );
  }
  getAllPosts() {
    return this.http.get('https://jsonplaceholder.typicode.com/posts', {
      context: new HttpContext().set(PASS_TOKEN, false),
    });
  }
  addUser(user: IUser, isAdmin: boolean) {
    return this.http.post<IUser>(
      'https://jsonplaceholder.typicode.com/users',
      user,
      { context: new HttpContext().set(IS_ADMIN, isAdmin) }
    );
  }
}
```

كما نلاحظ استخدمنا الـAPI ('https://jsonplaceholder.typicode.com/users)، والدالة POST لإضافة البيانات، اما الآن ننتقل إلى الخطوة التالية وهي تجهيز الـcomponent لاستقبال وعرض البيانات، لذلك اولاً لنضيف الـMarkup الخاص بإظهار الزين، كالتالي:

app.component.html ملف

```
<input type="button" value="Add User" (click)="AddNewUser()" />
<br />
<hr />
<input type="button" value="Add Admin User" (click)="AddNewAdminUser()" />
```

كما نلاحظ هالذين الزرين يُنفذان دالتين، ومن اسمهما يتضح مهمتهما، اما الآن لنكتب محتويات هذه الدالة، كالتالي:

http-body-admin.interceptor.ts

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from './services/api.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent implements OnInit {
  constructor(private api: ApiService) {}
  ngOnInit() {
    this.api.getAllGames().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
    this.api.getAllUsers().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
    this.api.getAllPosts().subscribe({
      next: (data) => console.log(data),
      error: (error) => console.log(error),
      complete: () => console.log('completed..'),
    });
  }
  AddNewUser() {
    this.api
      .addUser(
        {
          email: 'test@test.com',
          id: 22322,
          name: 'Saad',
          phone: '00998787667',
          username: 'dev',
          website: '',
        },
        false
      )
      .subscribe({
        next: (value) => console.log(value),
```

```
        error: (error) => console.log(error),
        complete: () => console.log('completed..'),
      });
  }
  AddNewAdminUser() {
    this.api
      .addUser(
        {
          email: 'test222@test.com',
          id: 666656666,
          name: 'Fahd',
          phone: '87677687',
          username: 'dev2',
          website: '',
        },
        true
      )
      .subscribe({
        next: (value) => console.log(value),
        error: (error) => console.log(error),
        complete: () => console.log('completed..'),
      });
  }
}
```

كما نلاحظ كلا الدالتين تستدعيان الدالة AddUser التي انشأناها سابقاً، ومن ثم تُرسلان نفس البيانات ولكن بقيم مختلفة كبارامتر أول لدالة AddUser، اما البارامتر الثاني فهو القيمة المنطقية حيث في إضافة مستخدم جديد تكون false اما في حالة إضافة مستخدم admin فتكون true.

اما الآن الخطوة الأخيرة فهي الذهاب إلى الـInterceptor ومن ثم إضافة الـLogic البرمجي اللازم لاعتراض هذا الاتصال ومن ثم إضافة وتعديل الـBody من عدمه، كالتالي:

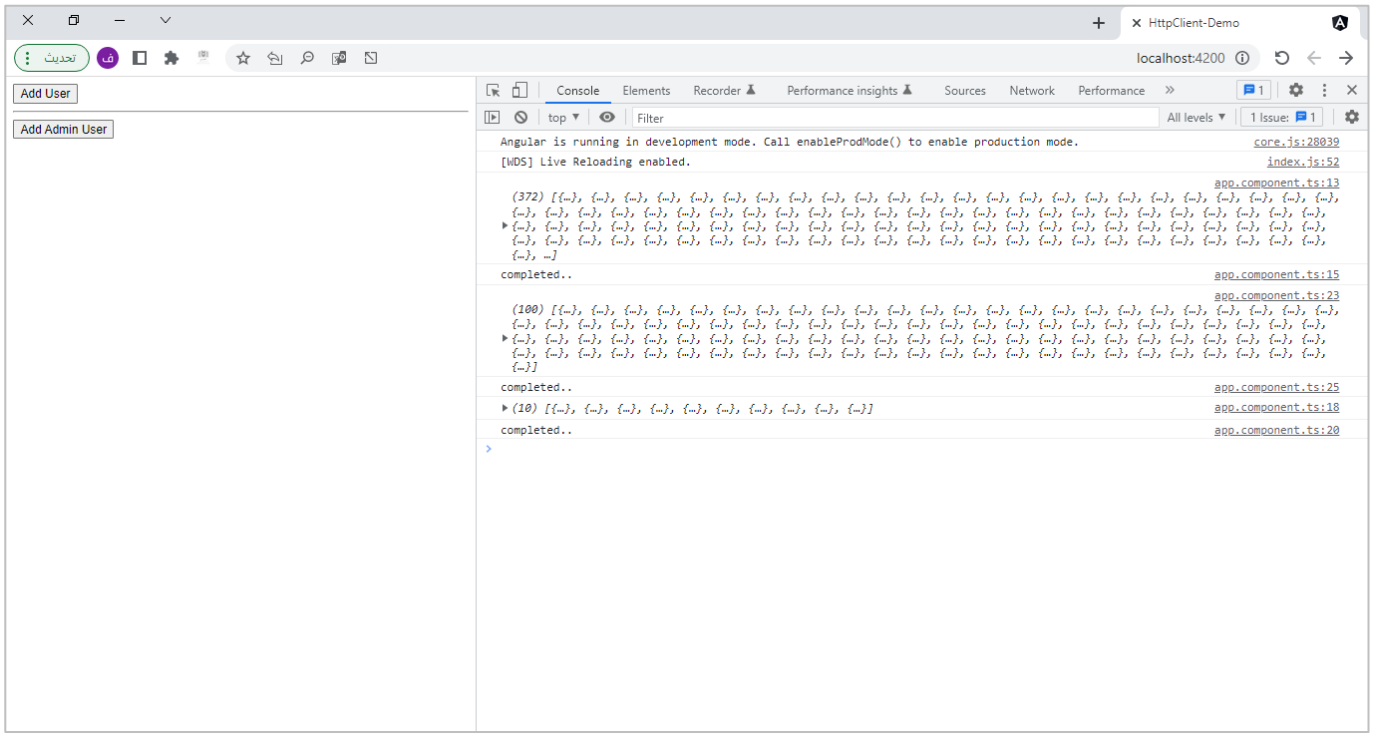
ملف http-body-admin.interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { IS_ADMIN } from '../shared';

@Injectable()
export class HttpBodyAdminInterceptor implements HttpInterceptor {
  constructor() {}

  intercept(
    request: HttpRequest<any>,
```

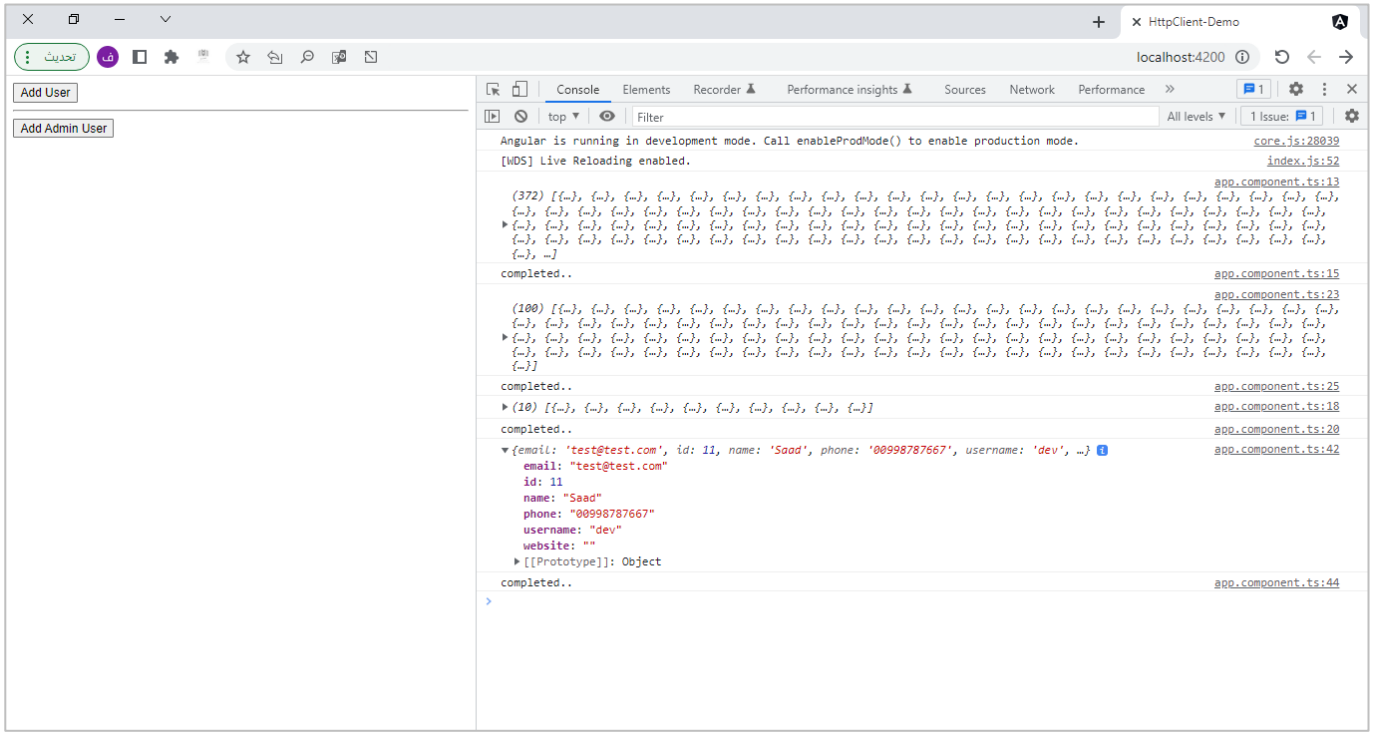
```
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    if (request.context.get(IS_ADMIN) === false) {
      return next.handle(request);
    } else {
      request = request.clone({
        body: { ...request.body, admin: 'admin' },
      });
      return next.handle(request);
    }
  }
}
```

كما هو مُلاحظ استخدمنا خاصية الbody لكي نستطيع إضافة او تعديل البيانات، ومن ثم وعن طريق ميزة Spread Operator (...) قمنا بدمج الBody الذي تم ارساله سابقاً مع إضافة البيانات الجديدة وهي القيمة admin.

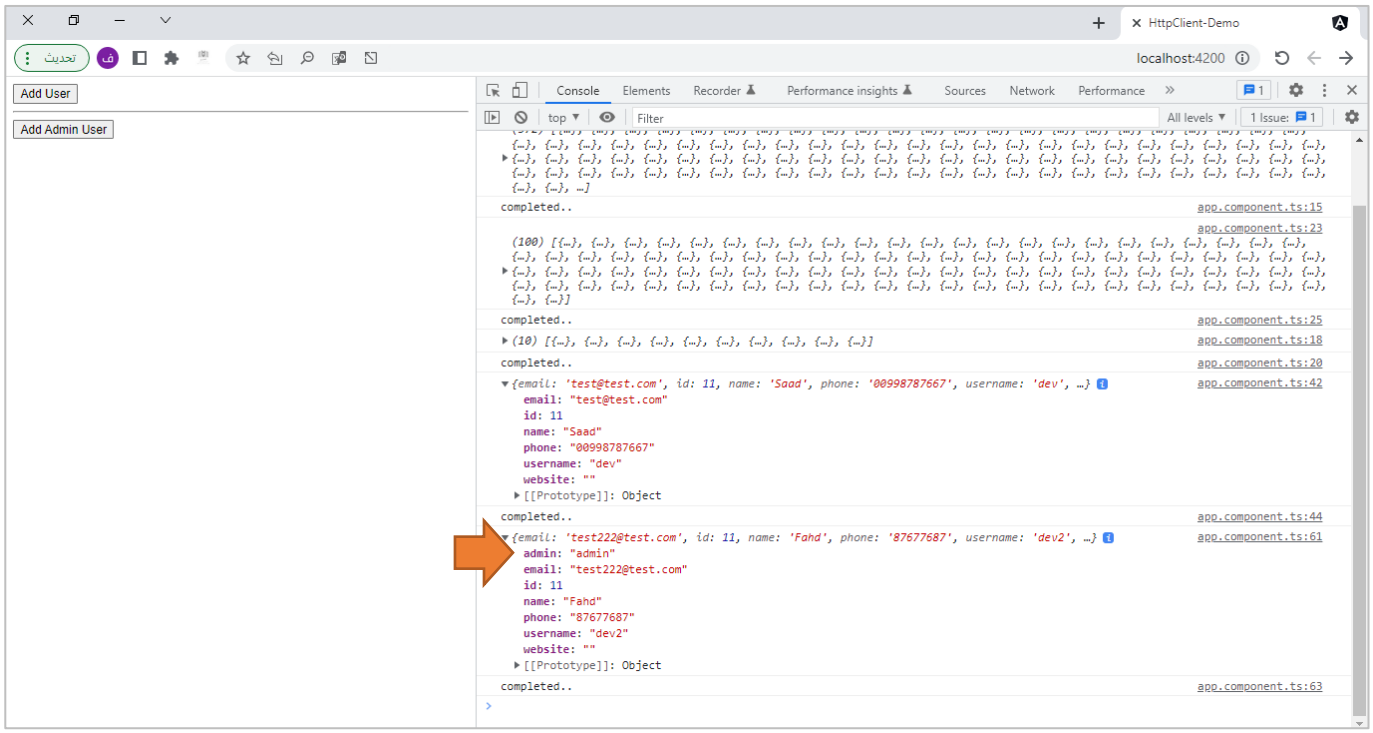
اما الآن لنقوم بمشاهدة النتيجة في المتصفح، كالتالي:



والآن لنضغط على زر Add User ولنرى النتيجة:



كما نلاحظ تم ارسال البيانات كما هي بدون أي تعديل من قبل الـInterceptor، اما الآن لنضغط على زر Add Admin User، كالتالي:



اما هنا فتم إضافة بيانات {admin: admin} إلى البيانات المُرسلة حيث تم اعتراض هذا الاتصال وإضافة هذه البيانات من قِبل الـInterceptor.

وبذلك تعلمنا كيفية الاستفادة من الـInterceptors لاعتراض الـBody للـRequest والتعديل عليه قبل ارساله إلى الخادم Server.

3-10- التعديل والتحكم بالResponse من خلال Interceptor:

كما نستطيع اعتراض Response القادم من الServer قبل وصوله إلى تطبيقنا بواسطة الInterceptors وإجراء أي تعديل نُريده عليها، كأن نقرأ الBody للResponse في حال وجود بيانات ونقوم بالتعديل عليها او نقرأ الHeader ايضاً للResponse وبناءً على خاصية فيها او مجموعة خصائص نقوم بإجراء امر معين.

وفي Angular Interceptors نستطيع التقاط الResponse من خلال دوال Pipe لمكتبة الRXJS، لذلك لنقوم بضرب مثال بسيط وذلك من خلال اجراء تعديل على الResponse للAPI الذي اعطينا مثال عليه سابقاً وهو (https://free-to-play-games-database.p.rapidapi.com/api/games)، وسوف نضع شرط بحيث لا يتم تنفيذ هذا الInterceptor إلا على هذا الرابط فقط، ومن ثم نستخدم دوال الpipe لكي نقرأ الResponse ومن ثم نقوم بتغيير الBody بحيث نقوم بفلترة النتائج وإرجاع قيمة معينة من جميع البيانات القادمة، كالتالي:

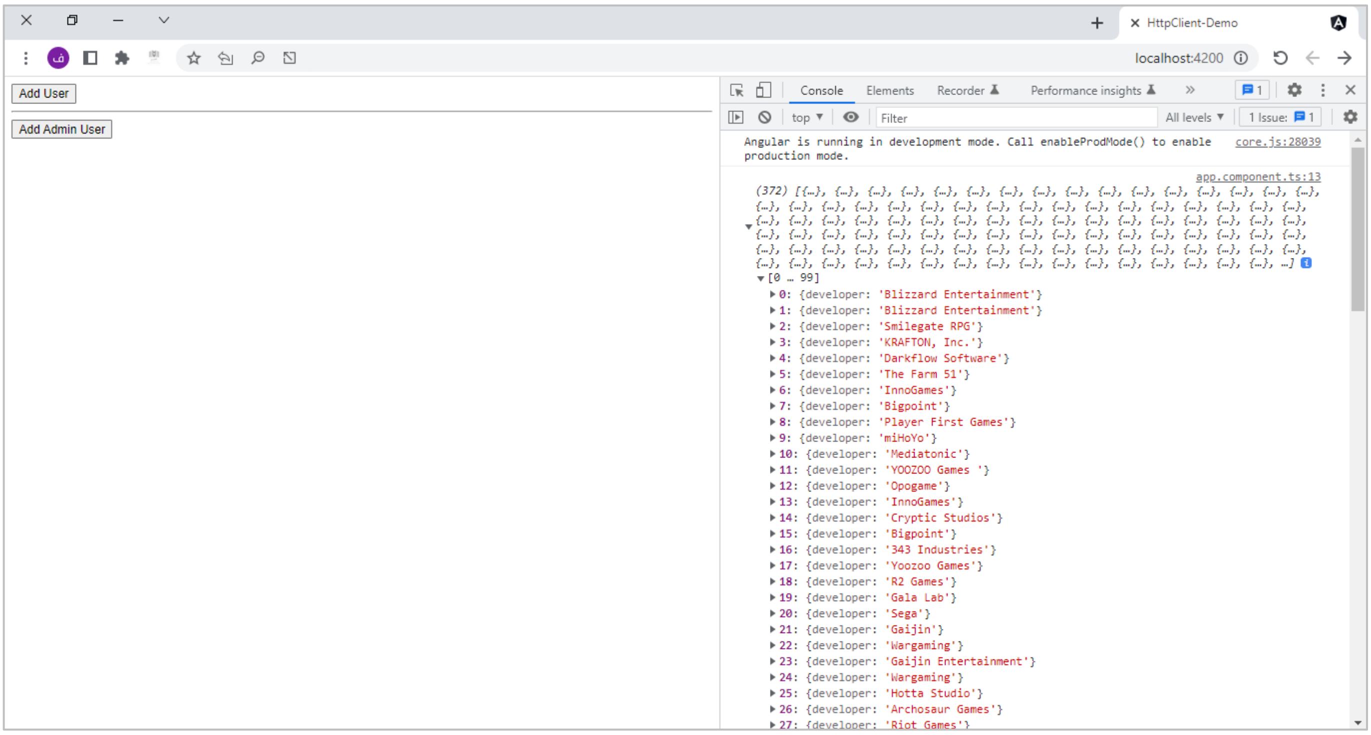
ملف http-header.interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
  HttpResponse,
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class HttpHeaderInterceptor implements HttpInterceptor {
  constructor() {}

  intercept(
    request: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    if (request.url === 'https://free-to-play-games-database.p.rapidapi.com/api/games')
{
      request = request.clone({
        setHeaders: {
          'X-RapidAPI-Key':
            'b52128808dmsh5826403ec30ac21p1b9548jsnfca5769e0b68',
          'X-RapidAPI-Host': 'free-to-play-games-database.p.rapidapi.com',
        },
      });
      return next.handle(request).pipe(
        map((event: any) => {
          if (event instanceof HttpResponse) {
            const data = (event.body as string[]).map((value: any) => {
              return { developer: value.developer };
            });
```

```
          return event.clone({
            body: data,
          });
        }
        return event;
      })
    );
  }
  return next.handle(request);
 }
}
```

كما هو واضح استخدمنا ميزة url لكي نتحقق من ان هذا الinterceptor لا يعترض إلا هذا الرابط فقط، والنقطة الأخرى المهمة هي عند استخدامنا لدوال Pipes حيث استخدمناها بعد الnext لكي نلتقط هذا الاتصال مع وضع شرط لنتأكد ان لا يتم قراءة الBody والتعديل عليه إلا في حالة كان نوع الاتصال هو Response، ومن ثم نقوم بإجراء التعديل اللازم على البيانات وإرجاع كائن يحتوي على الdevoloper فقط، ومن ثم نعمل لها clone واخيراً نقوم بإرجاعها بعمل return لكي يتم قرائتها وعرضها في الcomponent.

اما الآن لنشاهد النتيجة في المتصفح، كالتالي:



كما نلاحظ تم عرض فقط بيانات developer فقط وهذا هو المتوقع.

المراجع (References):

1. Freeman, **Adam, Pro Angular 9**, Apress, 2020.

2. Agarwal, Uttam, **Hands-On Full Stack Development with Angular 5 and Firebase**, Packt, 2018.

3. Delaney, Jeff, **The Angular Firebase Survival Guide**, leanpub, 2018.

4. Saha, Depasis, **Angular 7.0 for Beginners**, 2018.

5. Vijay, Santhosh, **Material for Angular Developer Course**, Leanpub, 2019.

6. Hussain, Asim, **Angular From Theory to Practice**, 2017.

7. D. Booth, Joseph, **Angular Succinctly**, Syncfusion, 2019.

8. Squad, Ninja, **Become a ninja with Angular**, 2019.

9. Steyer, Manfred, **Enterprise Angular**, Leanpub, 2020.

10. Clow, Mark, **Angular 5 Projects**, Apress 2018.

11. Nate Murray, Felipe Coury, Ari Lerner, and Carlos Taborda, **ng-book The Complete Guide to Angular**, Fullstack.io, 2018.

12. Bouchefra, Ahmed, **Practical Angular: Build your first web apps with Angular 8**, Leanpub, 2019.

13. Hajian, Majid, **Progressive Web Apps with Angular**, Apress, 2019.

14. Savkin, Victor, **Angular Router**, Leanpub, 2018.

15.**RxJs Full Course In Arabic**
(https://www.youtube.com/playlist?list=PL1ano0qwNuByzrFGCTutAuvJOS2dA6v5f)